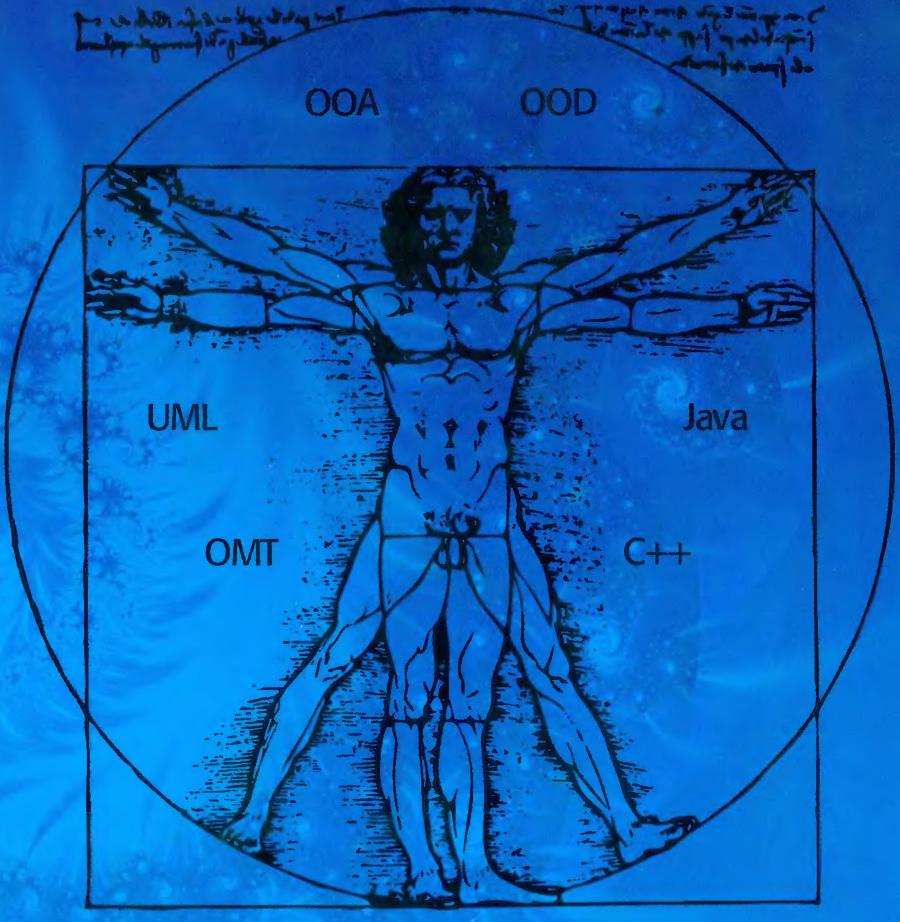
Vég Csaba

Alkalmazásfejlesztés

a Unified Modeling Language szabványos jelöléseivel



Logos 2000

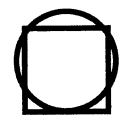
VÉG CSABA

ALKALMAZÁSFEJLESZTÉS

A UNIFIED MODELING LANGUAGE SZABVÁNYOS JELÖLÉSEIVEL

LEKTOR

dr. Juhász István



Logos 2000

1999.

© Vég Csaba, 1999

Első kiadás, 1999. május

Minden jog fenntartva. A szerző előzetes írásbeli engedélye nélkül jelen könyvet vagy annak részleteit más nyelvre lefordítani, valamint bármilyen formátumban vagy eszközzel reprodukálni, tárolni és közölni tilos.

A könyv, illetve a benne szereplő ismeretanyag felhasználásának következményeiért sem a Szerző, sem a Kiadó nem vállal felelősséget.

Az OMG az Object Management Group bejegyzett védjegye.
Az UML az Object Management Group védjegye.
Az Objectory a Rational Software Corporation bejegyzett védjegye.
A Rational Unified Process a Rational Software Corporation védjegye.
A Java a Sun Microsystems, Inc. bejegyzett védjegye.

A könyvben szereplő más védjegyek és bejegyzett védjegyek
nagy kezdőbetűvel vannak szedve.

ISBN: 963 03 7660 1

Kiadó: *Logos* 2000 Bt.
4032, Debrecen, Poroszlay 75.
logos2k@mail.westel.hu
www.westel.hu/~logos2k

Felelős kiadó: A *Logos* 2000 Bt. ügyvezető igazgatója

Borítóterv: Petromán László

Nyomdai munkák: Kapitális Bt., Debrecen

Bevezetés

Napjainkra az objektumorientált szemlélet az informatika egyik legjelentősebb tényezőjévé vált. Az „OO" rövidítés állandó jelzője lett a magas színvonalú szoftveralkalmazásoknak, amelyektől joggal várjuk el a nagyobb megbízhatóságot és flexibilitást. Egyrészről nyilvánvalóvá váltak a jól megtervezett OO alkalmazás előnyei, például a robosztusság, a fejlesztési ciklusok rövidülése és az egyszerűbb módosítás lehetősége, mely leegyszerűsíti a változások követését. Azonban be kell vallanunk, hogy több mint három évtizedes múltja ellenére az objektumorientált szemlélet még mindig nem érett és teljes. Legtöbb fogalma bizonytalan és homályos, és a háttérben sem áll egzakt matematikai elmélet. Az OO szemlélet így irányzatról irányzatra, sőt, személyről személyre más és más értelmet kap. Mindenki megalkotja a saját szabályait és konvencióit, melyeket több-kevesebb sikerrel alkalmaz a gyakorlatban.

Kialakulatlansága miatt ma az objektumorientált technológia alkalmazása személyekhez, „gurukhoz" kötődik, akik a napi munkában felbukkanó kérdőjelekre tapasztalatuk vagy éppen helyes ráérzésük alapján képesek megadni a válaszokat. A technológiai sor hiányosságaival és bizonytalanságával indokolható, hogy sokan még nem OO technológia mellett döntenek, vagy éppen elakadnak egy fejlesztés közepén.

Több szempontból is kísérleti művet tart a kezében az Olvasó. A következő oldalak egy rendhagyó alkalmazásfejlesztési módszert ismertetnek, amely célja, hogy az objektumorientált technológiát ipari méretekben is alkalmazhassuk. Ennek feltétele egy érthető, világos és ezért könnyen alkalmazható technológiai sor, hogy ne csak „guruk" legyenek képesek jó minőségű szoftver gyors előállítására. A módszer a „***Q***" elnevezést kapta, amely részben a minőségre *(„quality")* és a gyorsaságra *(„Quick modeling method")* utal. A „gyors" jelzőt olyan értelemben használjuk, hogy egyrészt a módszer segítségével csak a valóban szükséges lépéseket hajtjuk végre, így nem készítünk felesleges dokumentumokat vagy éppen programkódokat, csak azokat, amelyekre egy ipari méretekben előállított (tehát bonyolult, nem saját használatra készült és a későbbiekben valószínűleg módosítandó) alkalmazás esetén szükségesek. Másrészt a gyorsaság abból ered, hogy fejlesztés közben minimálisra csökkenthetjük a bizonytalanságot, azaz mindig tudjuk, hogy mi a következő lépés.

A módszer külön érdekessége, hogy nem kifejezetten OO szemléletű, hanem olyan módon használja fel az objektumorientált eszközkészletet, ahogy az OO a strukturált szemléletet. Ezen „új" szemlélet irányába történő lassú elmozdulás az informatika több területén is megfigyelhető, így elképzelhető, hogy a fejlődés következő mérföldköve hasonló jellegű koncepciókra fog épülni.

Jelen könyv a *Unified Modeling Language* (*„Egységesített modellező nyelv"*, rövidítve: UML) jelölésrendszerét használja fel a modellek ábrázolására. Minden jelölés tárgyalásra kerül, így a könyv első része UML tankönyvként is használható.

Az UML jelöléseivel az elkészített modelljeinket szabványos módon ábrázoljuk. Az UML-t a három legelterjedtebb objektumorientált módszer szerzője alkotta meg, amely tény már önmagában elegendő lenne ahhoz, hogy jelöléseik „de facto" szabvánnyá váljanak. Az UML mögé azonban több száz informatikai cég sorakozott fel, így javaslatuk eredményeként az UML-t közös (OMG-) szabványként fogadták el.

A jelölések és technikák mögötti implementációs lehetőségeket bemutató példák többnyire `Java` nyelven készültek. A `Java` nyelv valószínűleg a következő évek közös programozási nyelve lesz, egyfajta objektumorientált *`lingua franca`*, hasonlóan a szabványos modellező nyelvhez, az UML-hez. A fejlesztési módszer alapelveinek megértéséhez azonban nem szükséges sem a `Java`, sem más programozási nyelv ismerete.

El kell fogadnunk, hogy az informatika „anyanyelve" az angol. Egy program elkészítésekor mindig döntenünk kell, hogy a belső elnevezésekben és megjegyzésekben tiszta angol nyelvet, vagy egy bizonyos szintű keveréket alkalmazunk. Ha lehetséges, akkor célszerűbb a tiszta angolt választani, mivel a nyelv is egyszerűbb szerkezetű, a programozási nyelv kulcsszavait pedig úgysem fordíthatjuk magyarra. A könyv példáiban magyar alapneveket találhatunk, például `Alkalmazott`, vagy `Cég`. A többi elnevezés csak látszólag keverék, mivel a `hasAlkalmazott`, `everyAlkalmazott` stb. azonosítókban a `has` és az `every` prefix valójában kulcsszó.

Egyetlen módszer sem univerzális és kizárólagosságra sem törekedhet. Minden módszer egy javaslat, egy vezérfonal, melytől el lehet és el is kell térni, ha a fejlesztés adott lépése ezt sugallja vagy éppen ezt követeli meg. A szerző mindössze egy lehetséges fejlesztési módszer vázát mutatja be, amelyet az azt alkalmazó a saját követelményeinek megfelelően kiegészíthet és átalakíthat. A „***Q*** modellező módszer" sem egy lezárt koncepció, hanem alkalmazásának tapasztalatait feldolgozva az alapvető keretein belül folytonosan változik.

Köszönetet szeretnék mondani Juhász Istvánnak és Fazekas Ildikónak értékes megjegyzéseikért. A könyv nem jöhetett volna létre a Kossuth Lajos Tudományegyetem Matematikai és Informatikai Intézet, illetve az Információ Technológia tanszék segítsége nélkül. Köszönöm Németh Miklósnak (`IQSoft Rt.`) a gondolatgazdag közös eszmecseréket. Az összegyűlt tudásanyaghoz nagymértékben hozzájárultak a *Rendszerfejlesztés technológiája* előadáson aktívan részt vevő egyetemi hallgatók, valamint az `IQSoft Rt.` által rendezett *„OMT: Objektumorientált modellezés"*, illetve az *„UML – a szabványos objektumorientált modellező nyelv"* tanfolyamok résztvevői.

Tartalomjegyzék

Modellezés 1
Rendszerszervezés 1
Modellezés 2
Vizuális jelölések 3
Bonyolultság kezelése 4
A valóság modellezése 6
Absztrakt modellek 8

A történet 11
Programozási nyelvek 11
Módszertanok 13
Egységesítés és szabványosítás 15
Mi az UML? 17

Nyelv és módszer 19
Az OMT jellemzői 19
Rational Unified Process 20

A „Q" modellező módszer 22
A fejlesztés lépcsőfokai 23
Kiegészítő ajánlások és értelmezések 25
A szemlélet 25
Hogyan használjuk? 26
A „Q" modellező módszer és a RUP 26

I. KÖVETELMÉNYEK RÖGZÍTÉSE 29

Megjegyzések, fogalomszótár 30
Megjegyzések 30
Elnevezések 31
Fogalomszótár 32

Áttekintés 33

Kiterjesztési mechanizmusok 34
Sztereotípia 34
Megszorítások 36
Megszorítás és kulcsszavas érték 36
Kiterjesztési mechanizmusok használata 37

Használati esetek 38
Aktor 38
Használati eset 39
Aktor és használati eset 41

Aktorok általánosítása és pontosítása 42
Változatok (extends) 43
Részfunkció (include) 45
Használati esetek általánosítása 45
Változat, részfunkció vagy általánosítás? 46
Diagramok kialakítása 47
Felhasználó céljai és az interakciók 48
Használati eset diagramok 49

Csomagok 51
Csomag (package) 52
Függőségek 53
Névterület (namespace) 54
Csomag pontosítása 54
Csomagok használata 55

Rendszerek együttműködése 56
Közvetett vezérlés 59

A követelményelemzés szöveges dokumentumai 61
Alkalmazási példák 61
Forgatókönyvek 63
Működési leírások 65
Felhasználói felületek 65
Előzetes felhasználói leírás 68

Követelményelemzés 69
Kidolgozás 70
A követelményelemzés célja 71

II. OSZTÁLYDIAGRAMOK **73**

Osztály 74
Objektum és osztály 74
A szakterületi modell első elemei 76
Attribútumok 77
Műveletek 78
Lekérdezés és módosító 79
Származtatott attribútum és osztály 80
Láthatóság 80
Lekérdező és beállító műveletek 81
Attribútumok és műveletek értelmezése 82

Jellegzetességek 86
Részletezettség szintjei 86
Közös sztereotípia és megszorítás 87
Részek 87
Változatlanság 88
Jellegzetességek elérése 88

Jellegzetességek keresése **90**
Adatszolgáltató és tároló felelősségek 90
Entitások és értékek 91
Műveletek keresése 93
CRC-kártyák 93
Felelősségek és belső osztályok keresése 94
A művelet alanya és tárgya 97

Kommunikációs útvonalak **98**
Kommunikációs út 98
Asszociáció és kapcsolat 99
Szerepek 100
Attribútum és osztály számossága 101
Módosítás lehetősége 102
Elnevezések 102
Kollekció és felsorolás 103
Származtatott és részhalmaz asszociációk 104
Asszociációk értelmezése 105

Elérési utak **110**
Minősített asszociáció 110
„Vagy" asszociáció 111
Asszociációs osztály 112
Aggregáció és kompozíció 113
Erős és gyenge kapcsolat 115
Attribútum és asszociáció 116
Függőségek 117

Általánosítás és pontosítás **119**
Általánosítás és pontosítás 119
Többszörös osztályozás 121
Megkülönböztető 123
Dinamikus és többszörös változat 124
Kiterjesztés és korlátozás 125
Műveletek átdefiniálása 126
Osztályhierarchia 128
Absztrakt osztályok és interfészek 128
Generikus osztály 132

Különleges műveletek **134**
Értékek másolása és ellenőrzése 134
Konstrukció és destrukció 135
Kezdőértékek értelmezése 136
Konverzió 137

Kiegészítő technikák **138**
Osztályattribútumok és osztályműveletek 138
Metaosztály 140
Sztereotípia 141

Invariánsok, elő- és utófeltételek 141
Műveletek végrehajtási módja 142

Objektumok **144**
Objektumok 144
Objektum attribútumértékei és állapota 145
Objektumok és kapcsolatok 146

Osztálydiagramok használata **149**

III. INTERAKCIÓ-DIAGRAMOK **151**

Interakció-diagramok **152**
Példaobjektum 153
Üzenet 153

Szekvencia-diagram **155**
Konkurens folyamatok 156
Üzenetek 157

Együttműködési diagram **159**

Interakció-diagramok használata **161**
Tervezési minta 161

IV. IDŐBEN LEZAJLÓ VÁLTOZÁS **163**

Aktivitás-diagramok **164**
Aktivitás és sorrendiség 164
Akció- és aktivitás-állapot 167
Aktivitás és objektum 168
Szignál küldése és fogadása 171

Állapotdiagram **173**
Állapot és átmenet 173
Esemény 176
Esemény és átmenet 178
Állapot 178
Üzenetküldés 180
Tevékenység és akció 181
Strukturált állapotdiagram 182
Konkurencia 183

V. ADATFOLYAM-DIAGRAM **185**

Adatfolyam-diagram **186**
Feldolgozás és adatfolyam 187

Aktív és passzív objektumok 188
Vezérlésfolyam és strukturálás 189

VI. REPREZENTÁCIÓ **191**

Osztályként történő reprezentáció **192**
Felhasználói felületek 194

Pontosítás megvalósítása **196**
Pontosítás és kompozíció 198

Állapotdiagramok értelmezése **200**

VII. SZAKTERÜLETI ELEMZÉS **203**

Diagramok a követelményelemzésben **204**

Szakterületi elemzés **205**
Áttekintő szakterületi modell 206
Részletes szakterületi modell 209
Csomagok és osztályok 211

VIII. IMPLEMENTÁCIÓS DIAGRAMOK **213**

Komponens-diagramok **214**

Alkalmazási-diagramok **216**

IX. TERVEZÉS ÉS MEGVALÓSÍTÁS **219**

Alkalmazási környezet **220**

Részletes tervezés **222**

Megvalósítás **225**

A. „OBJEKTUMOKON TÚL" **227**

„Beyond objects" **228**
Létezik-e informatika az objektumokon túl? 232

Irodalomjegyzék **237**

Tárgymutató **239**

Modellezés

Bonyolult számítógépes rendszereket („alkalmazásokat") gyakorlatilag lehetetlen közvetlen programozással kifejleszteni. Ehelyett célszerű először a feladatot vázlatosan megfogalmazni, majd az így kapott tervet ismételt lépésekben egészen addig a szintig finomítani, amelyből egy automatikus eljárás már elő tudja állítani a gép által végrehajtható programot.

A gyakorlat bebizonyította, hogy bonyolult, több ember összehangolt munkáját igénylő nagyméretű szoftverrendszereket lehetetlen közvetlen kódolással, programírással kifejleszteni. A konkrét megvalósítás a feladat előzetes átgondolását igényli.

alkalmazásfejlesztés alapproblémája

A probléma az emberi figyelem korlátozott voltával és a számítógépes rendszerek sajátosságaival magyarázható. A pszichológiában már régen ismert tény, hogy az emberi elme egyszerre csak igen korlátozott számú elem viszonyát képes áttekinteni. Ezzel szemben a számítógépes alkalmazások általában nagyon bonyolultak, igen változatos és összetett viselkedést kell megvalósítaniuk, ugyanakkor ezt a bonyolultságot tökéletesen egyértelműen kell leírnunk. Egy étel elkészítése is bonyolult, azonban a szakácskönyvek receptjei csak vázlatosan rögzítik az „eljárást" — gondoljunk csak arra, hogy pontosan mennyi is egy „csipetnyi só". A jó szakácsnő vagy szakács a tapasztalatára és az ösztöneire hagyatkozva ki tudja egészíteni ezeket a hiányosságokat. A számítógépnek azonban nincs semmi tapasztalata és végső soron csak primitív, elemi gépi kódú utasítások végrehajtására képes.

Rendszerszervezés

vázlatos tervek

Nagyméretű alkalmazások fejlesztése esetén a siker csak úgy biztosítható, ha a tényleges programozás előtt elkészítjük a feladat vázlatos tervét. Ez ugyan (sem gépi, sem programozói szempontból) nem írja le egyértelműen az alkalmazást, de arról egy átfogó és — ami a leglényegesebb szempont — áttekinthető képet mutat. Ezután a vázlatos tervet ismételt lépésekben, újabb és újabb részletek meghatározásával addig finomítjuk, amíg elérjük az egzaktság szintjét.

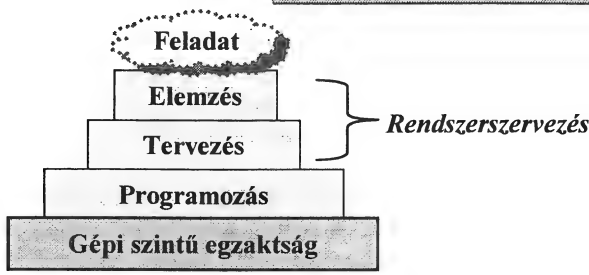
rendszerszervezés

Természetesen a végső célunk az egzaktság gépi szintjének az elérése, azaz egy vagy több programozási nyelven és környezeten történő megvalósítás. Ez azonban úgy is teljesíthető, ha a fejlesztők egy csoportja olyan részletezettségű dokumentumokat állít össze, amelyből a kiválasztott programozási nyelvekhez és környezetekhez értő másik csoport, a programozók („kódolók"), tudásukra és tapasztalataikra támaszkodva elő tudják állítani a gépi szintű egzaktságot. Az ilyen részletességű dokumentumok előállítását *rendszerszervezésnek* nevezzük. A helyzet hasonlít a főzésre: a „program-recepteknek" csak olyan fogalmakra szabad hivatkozniuk, amelyek a programozó számára egzaktak, és azokat át tudja ültetni a gép által már érthető nyelvezetre.

rendszerszervezés: elemzés és tervezés

A rendszerszervezés folyamatában célszerű további két lépést megkülönböztetni. Az *analízis* vagy *elemzés* (`analysis`) a feladatot bontja fel részeire, míg a *tervezés* (`design`) a már elemzett feladat számítógépes megvalósítását körvonalazza.

1. Ábra: Alkalmazásfejlesztés vázlata



A megfelelően gyakorlott „alkalmazástervező" az elemzés során részeire bontott feladatból kiindulva és a megvalósítással kapcsolatos stratégiai döntéseket (milyen programozási nyelvet, milyen adatbázis-kezelőt használjunk, stb.) figyelembe véve, ugyancsak önállóan („automatikusan") elő tudja állítani azt a részletesebb leírást, amelyet a programozók már meg tudnak valósítani. Az elemzők célja így a feladat olyan részletességű meghatározása, amely a tervezők számára már egyértelmű képet mutat.

Modellezés

modell

A különböző részletezettségi szintű vázlatos terveket az alkalmazás absztrakt modelljeinek, vagy röviden a *modelljeinek* nevezzük. A rendszerszervezésnek az a célja, hogy a tényleges kódolás előtt a feladat modelljeit állítsa elő.

modell készítésének előnyei

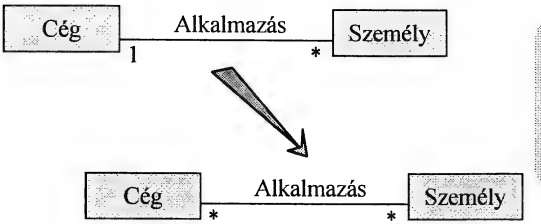
A modellek a programrészletekhez viszonyítva egy magasabb absztrakciós szinten (magyarul: vázlatosabban) ábrázolják az alkalmazást, így természetesen *áttekinthetőbbek*, mivel eltekintenek a konkrét megvalósítási technikák zavaró részleteitől. A munka- és időigényes programozás előtt előzetesen *ellenőrizhető*, hogy a terveink, mind a teljesség, mind a belső felépítés szempontjából valóban alkalmasak a feladat megvalósítására. Ha hibákat vagy hiányosságokat találunk, akkor a modellek a kész programoknál összehasonlíthatatlanul *könnyebben módosíthatók*: javíthatók, vagy éppen kiegészíthetők a hiányzó részekkel.

Egyes, az elemzés során elkészített vázlatos tervek a felhasználókkal és a *megrendelőkkel* történő *egyeztetésre* is alkalmasak, így ellenőrizhetjük, hogy valóban az igényeknek megfelelő alkalmazást fejlesztjük-e ki.

A következő ábrán szereplő, az UML jelöléseivel megrajzolt példánkban a `Cégek` és a `Személyek Alkalmazás` kapcsolatban vannak egymással. Az első változatban egy cég több személyt is alkalmazhat, ugyanakkor egy személy csak pontosan egy cég alkalmazásában lehet. Szükség esetén, például a megrendelő kérésére, egyetlen jel megváltoztatásával elérhetjük, hogy egy személyt ne csak egy, hanem több cég is alkalmazhasson, sőt, így a rendszerben olyan személy is

előfordulhat, aki egyetlen cég alkalmazásában sincs. Ez az „apró" módosítás a tervezésben, és különösen a programozásban igen lényeges változtatásokat igényel.

2. Ábra: Módosíthatóság



Megfelelő jelölésrendszer esetén egyetlen jel megváltoztatásával lehetővé tehetjük, hogy egy `Személyt` *ne csak egy, hanem több* `Cég` *is alkalmazhasson.*

Előzetes modellek segítségével a fejlesztés menete is jobban nyomon követhető, például pontosabban megbecsülhetjük annak erőforrásigényét, mind időben, mind a bekerülés költségében.

Vizuális jelölések

ábrázolás diagrammal

Áttekinthetőbbé és érthetőbbé tehetjük modelljeinket, ha ábrázolásukra grafikus jelölésrendszert, azaz diagramokat alkalmazunk. Diagramok segítségével a vázlatos terveket szó szerint „látványosan" jelenítjük meg. Modellünk „vizualizálásával" a feladatunk egyenként magas információtartalmú elemeit egy képileg is áttekinthető rendszerbe foglaljuk, így rendszerünk belső felépítését az alkalmazás *térképének*, vagy a ma divatos kifejezést használva: *tervrajzának* (`blueprint`) megrajzolásával szemléltethetjük.

alkalmazás térképe, tervrajza

3. Ábra: Diagramok

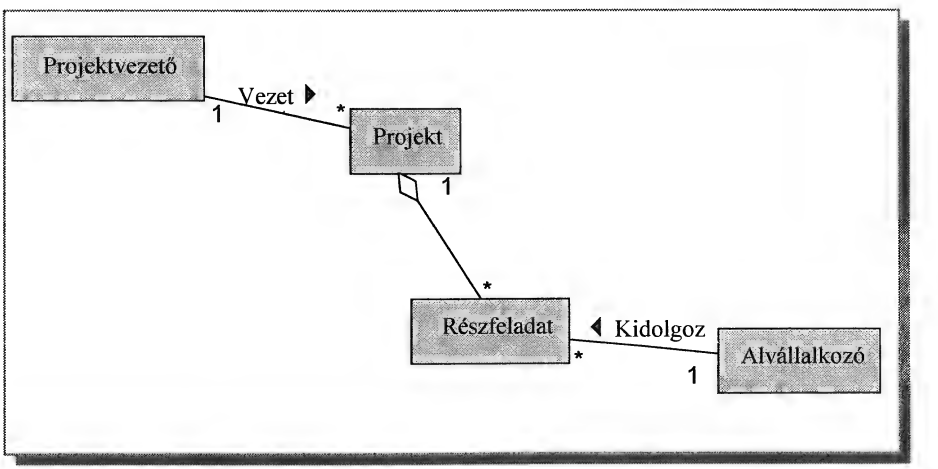


diagram: csomópontok és élek

Az alkalmazásfejlesztés során a feladatok ábrázolására használt diagramokon többnyire bizonyos *elemeket* és a közöttük lévő *viszonyokat* ábrázoljuk. Minden ilyen diagram matematikailag egy gráf, amely csomópontokból és az azokat összekötő élekből áll:

- a *csomópontokban* helyezzük el az alap és a kiegészítő elemeket,
- az elemek között húzott *élekkel* pedig a viszonyokat, kapcsolatokat jelöljük; az élek is bizonyos szempontból elemeknek („kapcsoló-elemeknek") tekinthetők, így közöttük is meghatározhatunk viszonyokat.

Minden egyes diagramot úgy jellemezhetünk, ha felsoroljuk, hogy milyen alap- és kiegészítő elemeket adhatunk meg a csomópontokban, illetve leírjuk, hogy élekkel a diagram mely elemei és milyen módon kapcsolhatók össze.

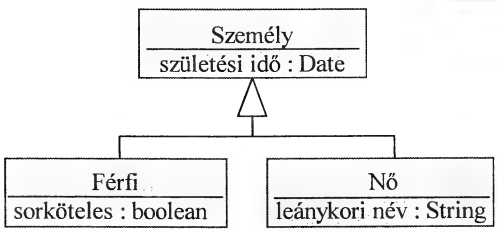
Bonyolultság kezelése

Napjainkra a *„bonyolultság kezelése"* (`dealing with complexity`) az informatika egyik központi problémájává vált. Az összetett rendszereket alapvetően két módszerrel lehet áttekinthetőbbé egyszerűsíteni.

absztrakció, részekre bontás

- *Az absztrakció* (szó szerint: elvonatkoztatás) segítségével eltekinthetünk a lényegtelennek ítélt részletektől, így a vázlatosabb kép csak a fontosnak tartott elemeket tartalmazza.
- *Részekre bontás* esetén a rendszert lazábban kapcsolódó elemekre osztjuk fel, mely részek (a kapcsolódási módok ismeretében) önállóan is vizsgálhatók.

4. Ábra: Absztrakció



Gyakran előfordul, hogy bizonyos elemek absztrakt képe megegyezik, mivel az elemek mind egy közös általánosnak a változatai. Ezek az elemek közös jellemzőkkel rendelkeznek, magyarul: *hasonlítanak* egymásra.

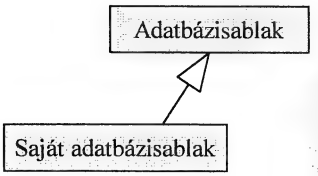
absztrakció, mint a kiemelés lehetősége

Ha a jelölés erre lehetőséget ad, akkor a hasonló elemek ismétlődő közös jellemzői az általános elemhez kiemelhetők. Ezért az absztrakció a *kiemelés* lehetőségét is jelenti, mely felhasználásával ugyanazt a rendszert számottevően rövidebben le tudjuk írni. Ugyancsak a kiemeléssel biztosítható a rendszer konzisztenciája, mivel ennek hiányában az általános elemen végrehajtott módosítást minden egyes változaton is végre kellene hajtani.

Ellentétes irányból tekintve az absztrakció a *megosztás* (`sharing`), más néven az *újrafelhasználás* (`reuse`) lehetőségeként jelenik meg, ami azt jelenti, hogy az egyszer megfogalmazott elemet az igényeinknek megfelelően specializálva a későbbiekben is felhasználhatjuk.

absztrakció mint az újrafelhasználás lehetősége

5. Ábra: Újrafelhasználás



Példánkban a beépített Adatbázisablakot a saját követelményeinknek megfelelően specializáljuk.

Az absztrakció egy érdekessége, hogy a jelölési rendszerek minősítésére is alkalmas. Így ugyanazt a feladatot rövidebben is megfogalmazhatjuk, ha a jelölések lehetőséget adnak a hasonló jellegű részletek magasabb szintű kiemelésére és egyetlen helyen történő megadására. A hagyományos szoftverekben egy-egy összefüggő adatcsoport manipulációi általában a program különböző helyein, szétszórva helyezkednek el. Az objektumorientált jelölések egyik nagy előnye, hogy az összefüggő adatcsoport (az „objektum") manipulációs lehetőségei kiemelhetők és az adatcsoporttal együtt, egyetlen helyen megadhatók.

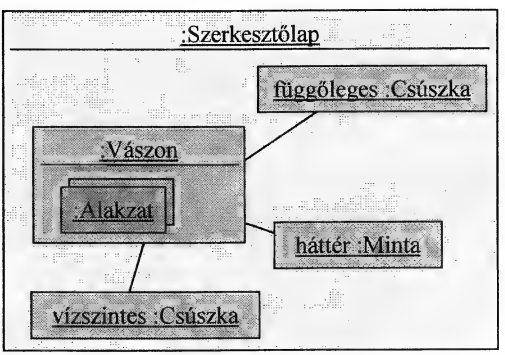
jelölési rendszerek minősítése

Ahogy az „egész" a részeit, úgy az absztrakció is az elemek egy csoportját határozza meg, de

- az absztrakció *„vagy"*-jellegű csoportosítást jelent, mivel egy absztrakt elem az „vagy" az egyik, „vagy" a másik változat,
- az egész a részeit pedig *„és"*-jellegű módon csoportosítja: az egész az egyik részéből „és" a másik részéből áll.

az absztrakció „vagy", az egész „és" jellegű csoportosítás

6. Ábra: Egész és részei



Mindkét technika csoportosítást határoz meg, ezért az általánosabb elemet és a több elemből felépülő, összetett „egész" elemet egyaránt *alapelemnek* (`base element, super element`), az általános változatait, illetve egy összetett elem részeit pedig *alelemeknek* (`subelement`) nevezzük.

alap- és alelem

példány

A modellezés szóhasználatában *konkrétnak*, *előfordulásnak* vagy *példánynak* `(instance)` nevezzük a *ténylegesen* megjelenő elemeket, például `Kiss Anna` egy konkrét `Személy`. Egy általános elem olyan keretet határoz meg, amelyet konkrét elemek tölthetnek fel tartalommal. Az általános így mindig lehetőségeket, azaz egy *lehetséges elemet* jelent. Konkrét elem létrehozását konkretizálásnak vagy *példányképzésnek* `(instantiation)` nevezzük.

absztrakt elem

Az *absztrakt* `(abstract)` jelzővel jelölt „tiszta" általános elem csupán lehetőségként létezik. Például egy alkalmazásban azt mondhatjuk, hogy általános `Alakzat` nem fordulhat elő, „csupán" adott alakzat, például `Téglalap`, `Ellipszis`, `Zárt poligon`, stb. Az általános, de nem absztrakt elem egy általános változatot határoz meg, amelyet az alváltozatok a megfelelő irányba tovább specializálhatnak (pl.: `Adatbázisablak` és `Saját adatbázisablak`).

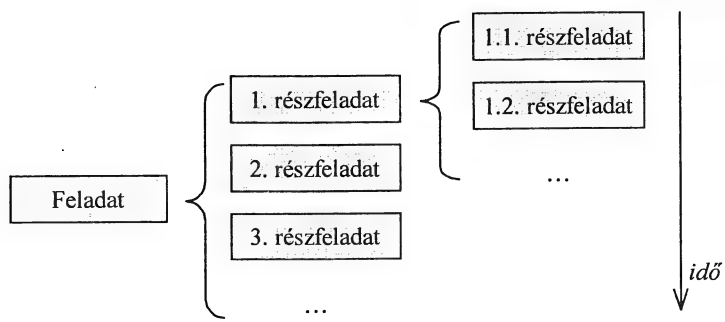
A valóság modellezése

A programozási, illetve a későbbi modellező nyelvek fejlődésének történetében egy határozott irányvonal mutatható ki. A jelölési rendszerek egyre inkább elszakadnak a konkrét gépi technikáktól és absztraktabbá válásukkal a valóság koncepcióinak ábrázolása kerül előtérbe.

strukturált módszerek

Az első jelentős minőségi ugrásnak a *strukturált programozási módszerek* megjelenése tekinthető. A számítógép mindössze elemi adatokon elemi gépi utasítások végrehajtására képes. Az összetett adat- és vezérlési szerkezetek segítségével azonban kifejezhetjük az adatelemeknek és programrészleteknek a valóságban is megfigyelhető szorosabb-gyengébb összetartozását, melyek csak közvetve jelennek meg a tényleges gépi kódban. Strukturált programtervezés esetén a teljes feladatot egyetlen absztrakt utasításnak tekintjük. Mivel a számítógép nem érti meg ezt a szintet, ezért azt konkretizálnunk kell úgy, hogy felbontjuk elemibb utasítások sorozatára, majd ezt a felbontást mindaddig ismételjük, amíg el nem érjük a gép által már érthető jelöléseket. A strukturált szemlélet ezért az időbeli sorrendiséget követve bontja fel részeire a folyamatot, az alapján, hogy mi történik először, majd azután, stb. A folyamat felbontásától *függetlenül* a szorosabban összetartozó elemek adatstruktúrákba foghatók össze.

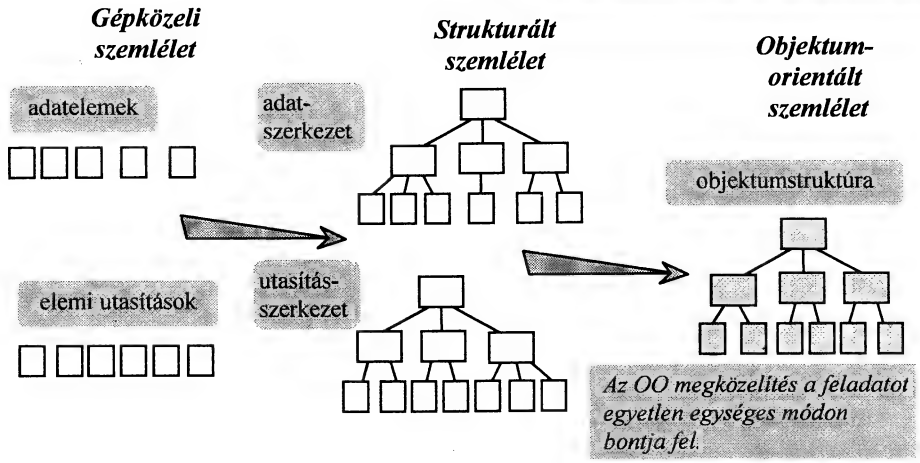
7. Ábra: Strukturált felbontás



objektum-orientált szemlélet

A fejlődés következő jelentős mérföldköve az *objektumorientált szemlélet*, amely a szorosabban összetartozó adat- *és* programelemek együttes megadására is rendelkezik eszközökkel. Egy hagyományos strukturált alkalmazásban egy adott adatcsoport kezelésével kapcsolatos utasítások általában a programban szétszórva helyezkednek el. Az OO jelölésekkel az adatokhoz kapcsolhatjuk az azt kezelő programrészeket, így bizonyos szinten egy önálló viselkedésűnek tekinthető objektumot kapunk. Képletesen: az adatszerkezetek „ismerik" felhasználásuk lehetséges módjait, így az adatoknak bizonyos „intelligenciát" kölcsönözhetünk. A szemlélet oldaláról közelítve az OO jelölések segítségével a valóság egy objektumának nem csak a tulajdonságait (az objektumra vonatkozó adatokat) ábrázoljuk, hanem azt is, hogy környezetének mely hatásaira képes reagálni, illetve milyen módon hathat a környezetére.

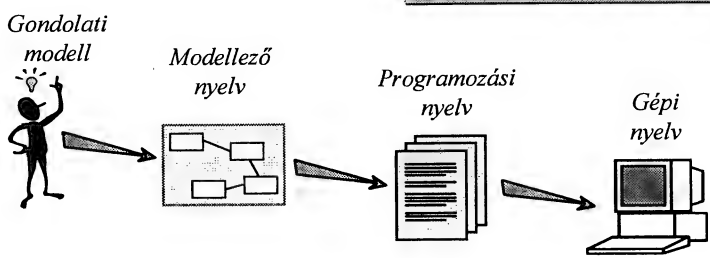
8. Ábra: Szemléletek fejlődése



valósághoz közelebbi jelölések

Ha egy jelölésrendszer távolabb áll a valóság koncepcióitól, akkor azon a tudatunkban megfogalmazott modelleket csak közvetve ábrázolhatjuk. A közvetettség azt jelenti, hogy a gondolati modellt több transzformációs lépésen keresztül át kell alakítanunk, míg a jelölésrendszer szintjét elérjük. A transzformációk segítségével szimuláljuk a valóság azon koncepcióit, amelyek a jelölésekkel közvetlenül nem ábrázolhatók.

9. Ábra: Modell transzformációja



hiányzó jelölések szimulációja

A szimuláció azonban legtöbbször azt jelenti, hogy adott technikát csak több jelölési elem kombinációjával tudunk ábrázolni, ezért ez a fajta szimuláció soha nem csökkenti, hanem általában jelentős mértékben megnöveli az ábrázolás méretét. Ellentétes irányba tekintve a valósághoz közelebbi jelölések használatával az *ábrázolásunk mérete* jelentős mértékben *csökkenhet*, mivel a minták helyett a gondolati koncepcióinkat „úgy, ahogy van", egyetlen jellel, közvetlenül is ábrázolhatjuk. A „jobb" jelölési rendszer ezért egyben absztrakciót is jelent, mivel kiemeli a jelölések adott módú használatát és ezért a rendszert is konzisztensebbé, egységesebbé teszi. Például az első számítógépek gépi kódjában nem szerepeltek a függvény hívásának és a hívásból való visszatérésnek az utasításai, így azokat a létező gépi utasításokkal kellett szimulálni. Gyakran előfordult, hogy még egyetlen programban sem ugyanazon a módon valósították meg az alprogramhívást és visszatérést, például más technikát alkalmaztak rekurzív és nem-rekurzív alprogramok esetén.

A valósághoz közelebbi, „természetesebb" jelölésekkel nem csak az ábrázolás mérete csökkenthető. Ha a gondolati modell megfogalmazásához kevesebb transzformációt kell végrehajtanunk, akkor a *modellünk egyszerűbben ábrázolható*, így a figyelmünket a feladatra (és nem az átalakítás módjára) összpontosíthatjuk. Egy kész modell vizsgálatakor a transzformációkat ellentétes irányba kell végrehajtanunk. A valóság koncepcióihoz közelebbi jelölésrendszer esetén ezért a kész *modellek áttekinthetőbbek* és *könnyebben értelmezhetők*.

természetesebb jelölésrendszer

A „valóság koncepcióin" mindig az *emberi tudat* valóságról alkotott koncepcióit értsük. „Természetesebbnek" is olyan jelöléseket nevezünk, amelyek közelebb állnak gondolkodási rendszerünkhöz. Az informatika így napjainkban egyre inkább a filozófiához, azon belül is az ismeretelmélethez, illetve az *általános rendszertudományhoz* (`General System Science`) és a *kibernetikához* közelít.

A probléma „mindössze" az, hogy nincsenek pontos fogalmaink arról, hogy a tudatunk milyen módon ábrázolja a valóságot. Pontosan emiatt egy kis elbizonytalanodás, megtorpanás figyelhető meg a jelölésrendszerek fejlődésében, különösen a jelölések jelentésének (szemantikájának), más néven a metamodelljeinek területén.

Másrészről viszont a modellezés és az információ ábrázolásának filozófiai-matematikai vizsgálatával felgyorsítható lenne ez a fejlődés, mely egyre természetesebb, azaz a mindennapos intenzív használat esetén könnyebben kezelhető jelölésrendszerekhez vezet.

Absztrakt modellek

„a mit és nem a hogyan"

Objektumorientált módszertanok esetén kiemelt szerepet kap a programozás előtti rendszerszervezés, így a kifejlesztendő alkalmazást az egymás utáni lépésekben finomított modelleken keresztül közelítjük meg. Az OO szemlélet szlogenjét követve „a *mit* és nem a *hogyan* a lényeges", azaz először azt fogalmazzuk meg, hogy milyen alkalmazást szeretnénk elkészíteni (a *„mit"*) és csak későbbre hagyjuk a megvalósítás technikai részleteit (a *„hogyant"*).

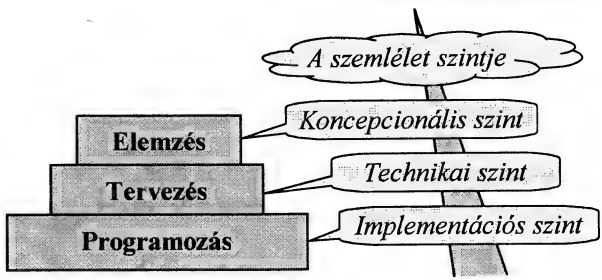
Eltérően a legtöbb hagyományos strukturált módszertantól, az objektumorientált megközelítések a modellek ábrázolására a rendszerfejlesztés minden fázisában azonos jelöléseket használnak. Ez egyrészt előny, mivel egyszerűbb az egyes fázisok közötti átmenet és a jelölésrendszer is könnyebben elsajátítható. Az azonos jelölések miatt azonban elmosódik a fázisok közötti határvonal, ez pedig könnyen egy gyakorlati problémához vezethet: egy OO rendszerfejlesztés során az elemzés gyakran tervezésbe vagy éppen implementációba „csúszik" át.

azonos jelölések a fejlesztés minden fázisában

Az objektumorientált személet egyszerre alkalmas a magasabb absztrakciós szinten álló, a valóság koncepcióihoz közelebbi modellek ábrázolására, ugyanakkor gépközeli megoldásokat is képes megjeleníteni. Az OO módszerek a hagyományos megközelítéseknél szélesebbre nyitják az absztrakt elvek és a konkrét megvalósítási technikák közötti tartományt. Ezzel a széles spektrummal magyarázható a szemléletnek az „emberi erőforrások" szempontjából vett legérdekesebb jellemzője, az, hogy elsajátítójától és alkalmazójától megköveteli, hogy egyszerre több absztrakciós szinten is gondolkodjon, illetve az éppen aktuális részproblémának megfelelő szintre váltson át.

szélesebb absztrakciós spektrum

10. Ábra: Megközelítési szintek



Melyek ezek az absztrakciós szintek? A módszerekkel egyrészt ábrázolhatjuk a készítendő alkalmazás logikáját, annak elvi részeit, a közöttük lévő kapcsolatokat és a változásokat. Ez az absztrakciós szint független a számítógépes fogalmaktól, így alkalmas bármely valós rendszernek, például egy üzleti folyamatnak az ábrázolására is. Ezt a szintet a továbbiakban *koncepcionális* szempontnak nevezzük, mely megközelítés a rendszerszervezés folyamatában az elemzésnek felel meg. A rendszerfejlesztés célja természetesen egy számítógépes alkalmazás létrehozása, ezért a koncepcionális modellünk minden egyes elemére meg kell határoznunk, hogy azt számítógépes környezetben milyen módon valósítjuk meg, azaz meg kell adnunk, hogy az elveket hogyan ültetjük át a gyakorlatba. Ezt a szintet a következőkben technológiai vagy röviden *technika* szintnek nevezzük, mely a rendszerfejlesztés folyamatában a tervezésnek felel meg. A konkrét, adott programozási nyelveken és környezetekben realizált megoldások alkotják az *implementációs* szintet. Például koncepcionális szinten meghatározhatjuk, hogy szükségünk lesz adott értékek rendezésére, a tervezés során kiválaszthatjuk a lehetséges rendezési technikák közül a gyorsrendezést, amelyet az implementáció szintjén leprogramozunk. Mindegyik absztrakciós szint a feladat vázlatosabb/elvi vagy részletesebb/technikai *nézetét* (`view`) adja meg.

koncepcionális szint

technikai szint

implementációs szint

Egy koncepcionális elem számítógépes környezetben történő megvalósításakor több lehetséges technika közül is választhatunk. Általánosan fogalmazva: a magasabb absztrakciós szint minden egyes eleme az alacsonyabb szinten lehetséges megoldási módokat határoz meg, azaz egy *absztrakt megoldási módot* jelöl ki. Az OO szemlélet elsajátításának és használatának nehézségét ezért úgy fogalmazhatjuk meg, hogy itt a fejlesztés során mindig az absztrakciós szintek teljes spektrumában kell gondolkodni. A jelölésekkel valós folyamatok önállóan is modellezhetők, illetve megérthetők, de ha azt is el szeretnénk képzelni, hogy az absztrakt fogalmakra épülő modellből hogyan lesz működő alkalmazás („hogyan lesz ebből kód"), akkor nem tehetjük meg, hogy a lehetséges megvalósítási technikáknak legalább az elvével ne legyünk tisztában.

absztrakt megoldási mód

Önmagában az eszközök (programozási nyelvek, stb.) ismerete sem elegendő, hiszen azok egy-egy lehetséges befogadó környezetként csak a kész terv számítógépes megvalósítására alkalmasak, a terv elkészítéséhez azonban nem adnak segítséget. A modellező nyelv és az eszközök viszonya hasonlítható a magasszintű programozási nyelvek és a gépi kód viszonyához.

Egy magasabb absztrakciós szinten álló modell realizálásakor az absztrakt megoldásokhoz egy-egy konkrét technikát kell rendelnünk. Ezt a hozzárendelést *leképezésnek* (`mapping`) nevezzük. Az objektumorientált jelölések tárgyalása során minden elemre megadjuk, hogy azok az egyes szinteken mit jelentenek.

modell leképezése

A modellek készítésének három szempontja felett létezik még egy absztrakciós szint, amely megadja, hogy az egyes jelöléseket milyen esetekben lehet, illetve célszerű használni. Ez a filozófiához közeli szempont a jelölésrendszert és a valóságot (pontosabban a róla kialakított képünket) kapcsolja össze, ezért ez a jelölések mögöttes tartalmát, a szemléletet, vagy más néven: a metamodellt jelenti.

A történet

Az objektumorientált koncepciók először különféle programozási nyelvekben jelentek meg, majd csak ezután alkalmazták a szemléletet a tervezésben és végül az elemzésben. A '90-es évek közepén az OO módszerek robbanásszerű megjelenésével az egységesítés és szabványosítás igénye is megszületett. A három legnépszerűbb módszertan megalkotója egy közös jelölési rendszert alakított ki, amelyet Egységesített modellező nyelvnek (UML) neveztek el. A legfontosabb informatikai cégeket is magában foglaló UML Partners támogatásával '97 novemberében az UML-t hivatalosan is szabványként fogadták el.

Az objektumorientált szemlélet történetének egyik érdekessége, hogy a legalapvetőbb koncepciók már a programozás fejlődésének első éveiben is megjelentek, elsősorban a mesterséges intelligenciával kapcsolatos keretrendszerekben.

Programozási nyelvek

SIMULA '67

Az első, ténylegesen objektumorientáltnak nevezhető általános célú programozási nyelv a `SIMULA '67` volt, amelyet arra terveztek, hogy valós folyamatokat diszkrét események sorozatával modellezzenek. A Dahl és Nygaard munkája során megszülető ALGOL-jellegű nyelv rendelkezett az osztályok, az egyszeres öröklődés, a művelet-átdefiniálás *(overloading)*, a statikus és dinamikus kötés (!) jelölésével, beépített automatikus tárfelszabadítással *(garbage collection)*, a párhuzamos működést lehetővé tevő korutinokkal és a szimulációt segítő kiegészítő osztályokkal.

Smalltalk

Az első tisztán OO nyelv a `Smalltalk` volt, amelyet a'70-es években a `XEROX Palo Alto` kutatóközpontjában fejlesztettek ki Alan Kay és Adele Goldberg vezetésével. Az „objektumorientált" jelzőt is először Alan Kay használta a `Smalltalk` jellemzésére, amellyel arra akart utalni, hogy a számítógép vezérlését nem elemi utasítások sorozataként adjuk meg, hanem az objektumok közötti üzenetváltások segítségével. A `Smalltalk` bevallottan a `SIMULA '67` nyelv koncepcióira épült és a `LISP` alapfelépítését örökölte.

A `Smalltalk` egyik újdonsága az üzenetváltás jelölése volt. Tisztán OO jellege is innen ered, mivel ezen a nyelven lehetetlen hagyományos módon programozni: csak osztályokat definiálhatunk, a működést pedig kizárólag az objektumok közötti üzenetváltásokként adhatjuk meg. Másik újdonság a grafikus felhasználói felületek *(Graphical User Interface – GUI)* megjelenése volt, amelyek fejlődése később is összefonódott az OO technológiával.

A `Smalltalk` több hátránnyal is rendelkezett, amely megakadályozta széleskörű elterjedését és ipari méretű alkalmazását. Jelölései — enyhén szólva —

szokatlanok, ezoterikusak voltak. A legkomolyabb problémák azonban a hatékonysággal kapcsolatban merültek fel.

C++

A problémákra a '80-as évek elején a C nyelv kiterjesztéseként megjelenő Objective-C, valamint az először C with classes, majd C++ néven elnevezett nyelv jelentette a megoldást. Alkotói a C++ programozási nyelvet a C olyan kiegészítéseként tervezték, amely szimulációs feladatok hatékony kezelésére alkalmas, így több nyelvi elemet is a SIMULA '67-ből vettek át. Ettől eltérő irányt jelentett az Objective-C, ami alapvetően a Smalltalk nyelvre épült.

A C++ nyelvet felhasználva az objektumorientált koncepciókat gépközeli szinten és hatékonysággal valósíthatjuk meg, így a C++ felnyitotta az OO technikák ipari méretű felhasználásának kapuit.

Az OO technikák gépközeli megvalósítása miatt a C++ egy igen bonyolult nyelv, amelynek teljes eszközrendszerét, a gyakorlati alkalmazás lehetőségeit és korlátait csak magasan kvalifikált programozók ismerik. Jellegénél fogva a C++ szinte automatikusan a leghatékonyabb és ezért nagyon magas költségű megoldást kényszeríti ki és majdnem lehetetlenné teszi, hogy a problémát nagyvonalúan, egy magasabb absztrakciós szinten közelítsük meg. Hasonló hatékonysági okokra vezethető vissza, hogy a C++ nem rendelkezik olyan beépített automatizmusokkal (például automatikus tárfelszabadítással) és védelmi eszközökkel (pl.: indexhatár-ellenőrzés), amelyek megkönnyítenék a biztonságos programok fejlesztését. Viszonylag kevés az olyan szabványos kiegészítése, amely szervesen a nyelvhez illeszkedik és segíti a tipikus feladatok megoldását, például az összetett adatok vagy éppen a felhasználói felületek kezelését. Egy C++ projekt sikerének biztosítása nagy tapasztalatot és komoly előkészületeket igényel.

Java

Az eredetileg a Wold Wide Web-dokumentumok egyszerű kiegészítéseként megszülető Java nyelv napjainkban a legesélyesebb jelölt a C++ felsorolt hiányosságainak kiküszöbölésére. A Java nyelv alapja egy egyszerűsített C++, amelyből száműztek minden nehezebben érthető konstrukciót, vagy amely megengedné a programozónak a „varázslás" lehetőségét, azaz, hogy olyan bonyolult módon valósítsa meg a feladatot, amely megnehezíti a programkód megértését.

Szemben a C++-al, a Java nyelv nem a hatékonyságot helyezte előtérbe, hanem az általánosságot (például a metódusok virtuálisak, minden objektumról megállapíthatjuk, hogy az egy adott osztály példánya-e, stb.), a belső automatizmusokat (pl.: automatikus tárfelszabadítás) és a biztonságosságot (pl.: figyeli, hogy nem léptük-e túl az indexhatárt, vagy nem akarunk-e egy üres mutatón keresztül hivatkozni egy területre). Beépített mechanizmusai miatt a Java nyelv esetén csak nehezen tehetjük meg, hogy ne nagyvonalúan közelítsük meg a feladatot.

Ugyancsak lényeges szempont, hogy a Java nem csak egy csupasz nyelvet jelent, hanem ahhoz egy igen nagy méretű osztálykönyvtár is tartozik. A nyelv és a környezet is elméletileg platform-független: a szlogen szerint az egyszer megírt és lefordított kód bárhol futtatható, ahol elérhető a Java virtuális gép. A Java így egy igen produktív környezet, amely lényegesen felgyorsíthatja az alkalmazások fejlesztését.

A Java mögé nagyon sok informatikai cég felsorakozott, így a nyelv és a környezet egyfajta objektumorientált `lingua franca`-ként („közös nyelvként") funkcionálhat a következő években.

Természetesen a Java is rendelkezik hiányosságokkal, így például a nyelvben is találhatók következetlenségek, nem megoldottak a sebesség és hatékonyság problémák, csak részben megoldott az objektumok külső tárolásának kérdése (a perzisztencia, illetve a relációs adatbázisban történő tárolás) és az alkalmazásoknak is igen nagy az erőforrásigénye.

A „tökéletes objektumorientált nyelv" kérdése tehát még nyitott. A Java azonban viszonylagos egyszerűsége és produktivitása miatt igen alkalmas az OO rendszerek megvalósítására.

Módszertanok

A '70-es évektől kezdve nyilvánvalóvá vált, hogy a nagyméretű és igen bonyolult rendszereket lehetetlen esetleges módon kifejleszteni. A jelenségre, amelyet szoftverkrízisnek neveztek el, a sikertelen fejlesztések tömege hívta fel a figyelmet. Bonyolult szoftverek esetén csak úgy biztosítható a siker, ha a barkácsolást ipari méretű fejlesztés váltja fel. Ehhez egy technológiai sorra, azaz egy módszerre vagy más néven eljárásra (processzre) van szükség, amely az esetlegesség helyett egy határozott irányvonallat képes mutatni. A jó alkalmazásfejlesztési módszer csökkenti az emberi tényező bizonytalanságát, így a fejlesztést követhetővé, ellenőrizhetővé és becsülhetővé teszi.

rendszer-fejlesztési módszerek

Ahogy az objektumorientált koncepciók megjelentek egyes programozási nyelvekben, elkezdték vizsgálni, hogy ez az új szemlélet hogyan alkalmazható rendszerfejlesztési módszerként. A kezdeti kísérletek csak fokozatosan szakadtak el a konkrét nyelvi elemektől. Először az objektumorientált tervezés *(OO design)* született meg, ami alapvetően a programok elkészítésére ad tanácsokat. Az elemzés *(OO analysis)* a kifejlesztendő alkalmazást már nyelvtől és platformtól független módon közelíti meg.

OO módszerek

A történet egy érdekessége, hogy az objektumorientált elemzés és tervezés kezdetei az `Ada` nyelvhez kötődnek. Ahogy a több évig csak specifikációjában létező `Ada` nyelv megvalósult, felmerült az igény a nyelv oktatására. Grady Booch '86-ban másodmagával kapta a feladatot, hogy állítson össze egy, az `Ada` nyelvet ismertető tanfolyamot.

Booch észrevette, hogy egyetlen hét még arra is kevés, hogy a nyelvi elemeket bemutassa, így tematikájában inkább arra szerette volna helyezni a hangsúlyt, hogy egy `Ada` programot milyen módszerrel lehet kifejleszteni. A szakirodalomban Abbot '83-ban publikált cikkére talált rá, aki kiindulópontnak a feladat szöveges megfogalmazásának egyszerű nyelvtani elemzését ajánlotta. A technika alapján például a főnevek lehetséges objektumokra, az igék pedig lehetséges műveletekre utalnak (pl.: „az árut a címre *továbbítják*"). Booch a hagyományos strukturált módszerek, illetve

Booch

az Intel cég kísérleti OO processzorának tervei mintájára a program alapszerkezetét diagramokkal ábrázolta. Az Intel ezen, `iAPX-432` nevű processzorának terveiből származtak az objektumokat reprezentáló híres-hírhedt felhőcskék *(blob)*, amelyek a későbbiekben sok kritikát kaptak, mivel mind kézzel mind pedig CASE eszközökkel (szoftverfejlesztést segítő programokkal) nehezen rajzolhatók voltak.

Az úttörő Booch módszerének látványos sikerét követve több tervezési módszer is született. Booch '91-ben átdolgozta jelöléseit és módszerét, amely már több elemzési elemet is tartalmazott és már nem csak az `Ada` nyelvre volt alkalmazható.

CRC-kártyák

Az elemzésre és tervezésre egyaránt használható a CRC-kártyák módszere, amelyet '89-ben Cunningham és Beck publikált. A CRC az osztály, felelősség és együttműködés *(class, responsibility, collaboration)* szavak rövidítése. Rebecca Wirfs-Brock '90-ben megjelent könyvében részletesen is tárgyalta a felelősségeken alapuló tervezésnek *(RDD: responsibility-driven design)* is nevezett technikát, amely megkönnyíti az alkalmazás osztályainak és azok felelősségeinek (attribútumainak és műveleteinek) azonosítását, illetve a felelősség megvalósításához szükséges további osztályok (az együttműködők) megtalálását.

RDD

OO elemzési módszerek

Az OO elemzési technikák fokozatosan nőttek ki a hagyományos strukturált elemzési módszerekből. Több szerző kiegészítette, illetve a későbbiekben átalakította technikáját az OO koncepcióknak megfelelően. Közöttük az első a Shlaer-Mellor szerzőpáros volt, akik '89-ben megjelent könyvükben az adatmodellezésnek egy objektum-alapú kiegészítését mutatták be. A Coad-Yourdon szerzőpáros elemzési módszerüket a '90-ben, illetve '91-ben megjelent könyveikben ismertették.

Rumbaugh, OMT

James Rumbaugh és négy társa (Blaha, Premerlani, Eddy és Lorensen) '91-ben publikálták az általuk összeállított jelölésrendszert és módszert, amelyet *Objektum-modellezési technikának (Object Modeling Technique – OMT)* neveztek el. A General Electric kutatóközpontjában megszületett OMT az egyik legkedveltebb módszerré vált.

Példátlan népszerűségét több okra is visszavezethetjük. Az OMT-t ismertető könyv közérthető nyelven íródott, így elsajátításához nem szükségesek mély informatikai ismeretek. Minden fogalmat, igaz csak informálisan (szövegesen), de egyszerűen és érthetően definiáltak. A legnépszerűbb jelölésekből kiindulva konzisztens jelölésrendszert állítottak össze. Módszerük pedig, mely alapja az Abbot által ismertetett szövegelemzés, egyszerűen, szinte gépiesen végrehajtható lépésekből állt.

Az OMT könyvet olvasva az embernek az az érzése, hogy megtalálta a „bölcsek kövét", mellyel bármely feladatot egy-két pillanat alatt meg tud oldani. A valóság, természetesen, ennél bonyolultabb. Az OMT gyakorlati alkalmazása a jelölések és a technika több gyenge pontját és hiányosságait is a felszínre hozta. Az OMT azonban egy áttekinthető és biztonságos alapot rakott le, így például használható az OO technikákkal való ismerkedés első lépéseként is. Az sem véletlen, hogy az UML majdnem teljes mértékben átvette az OMT jelöléseit.

Az objektumorientált módszerek általános hiányossága volt, hogy kevés támpontot adtak a követelmények tisztázására. Például az OMT az elemzés kezdetén adottnak tételezte fel, hogy a követelményeket már szöveges formában rögzítettük. A problémára a svéd Ericsson-nál a telefonos kapcsolórendszer szoftverein dolgozó Jacobson és társai által megalkotott Objectory (a név eredete: `Object FactOry`), illetve annak egyszerűsített változata az OOSE `(Object-Oriented Software Engineering)` adta meg a választ. A gyakorlatias megközelítésű OOSE legtöbb eleme más OO módszerekben is megtalálható, nagy újdonsága azonban a *használati esetek* `(use case)` technikája. A használati eset egyszerűen a rendszer és a felhasználó közötti tipikus interakció-sorozatnak egy rövid elnevezése. Ez az egyszerű technika lehetővé teszi, hogy összefoglaljuk a rendszerrel kapcsolatos követelményeket, a rendszer vázlatos külső „képét", így megrajzoljuk a rendszer határait.

Jacobson, Objectory, OOSE

használati esetek

A használati esetek technikája rövid idő alatt nagyon népszerűvé vált, amely különösen azért figyelemre méltó, mert bírálói ezt a módszert nem tartották objektumorientáltnak. Több módszer is ezt javasolta az alkalmazás fejlesztésének első lépéseként, például az OMT második változata, az OMT-II.

Az OO szemlélet kialakulatlanságára utal többek között az a tény, hogy fogalomrendszere még nem egyértelmű. Érdemes ebből a szempontból megemlíteni Odell és Martin '94-es könyvét, amely jelentős lépéseket tett a fogalmak tisztázására és a szemlélet metamodelljének megalapozására.

Odell és Martin

Egységesítés és szabványosítás

A kilencvenes évek közepére az objektumorientált technikák nagyon népszerűekké váltak. Sok csoport önálló jelöléseket és módszereket fejlesztett ki, így ezek száma körülbelül a tíz és az ötven között váltakozott. A nagyszámú módszer azonban inkább megnehezítette az OO technológia ipari méretű alkalmazását, mivel igen nehéz volt kiválasztani közülük az éppen megfelelőt.

A sok jelölés és módszer azonos koncepciókat használt, így természetesen mindegyikben megjelent az objektum, az osztály, a kapcsolat, az általánosítás, stb. fogalma. A változatok még megjelenésükben és módszereikben is hasonlítottak egymáshoz, a különbségek pedig néha csak az egészen elhanyagolható részletekben jelentek meg. Tovább bonyolította a helyzetet, hogy a jelentősebb módszereknek megjelent a második generációja (például a Booch '93 és az OMT-II), amelyek több koncepciót, jelölést és technikát is átvettek versenytársaiktól.

módszerek hasonlósága

A módszerek palettája tehát színes volt, sőt, túl színes. Az egyeztetésre több kísérlet is történt. Például az OO technikák szabványosítását irányító, több száz informatikai céget magában foglaló OMG `(Object Management Group)` egy csoportja többször is megpróbált összeállítani egy szabványos jelölést, Booch pedig informális megbeszéléseken („kötetlen reggeli kávézás közben") tett kísérletet az egyeztetésre — szintén sikertelenül. Minden módszertan megalkotója ragaszkodott a jelöléseihez, így ezt az időszakot gyakran a „módszerek háborújaként" `(method war)` is jellemezték.

„módszerek háborúja"

Az egyeztetés csak '94 őszén, az Egyesült Államok objektumorientáltsággal foglalkozó legnagyobb konferenciáján, az OOPSLA-n *(Object-Oriented Programming, Systems, Languages and Applications)* kezdődött el. Grady Booch, a Booch módszer atyja és James Rumbaugh, az OMT-t összeállító csoport vezetője közösen jelentették be, hogy *„A módszerek háborújának vége — mi nyertünk" (Method war is over – we won)*. Technikailag ez azt jelentette, hogy Rumbaugh csatlakozik Booch cégéhez, a Rational-hez, ahol együtt egy közös módszert dolgoznak ki.

Unified Method 0.8

Természetesen nagy várakozás kísérte kettejük munkáját. A '95-ös OOPSLA konferencián mutatták be az Egységesített módszert *(Unified Method)*, amelyet a 0.8 verziószámmal jelöltek arra utalva, hogy az 1.0 lesz majd a szabványosításra alkalmas és ezért ipari méretekben is alkalmazható változat. A „módszer" a Booch '93 és az OMT-II jelöléseit egységesítette, így a Jacobson féle használati esetek *(use case)* jelöléseit is tartalmazta. Az UM 0.8 mellett a másik meglepetés az a bejelentés volt, mely szerint Jacobson is csatlakozik (Objectory nevű cégével együtt) a Rational-hez és — ahogy elnevezték magukat — a „három amigo" együtt folytatja a munkát.

UML 0.9

Az UM sok kritikát kapott, mivel nem tartalmazott tényleges módszertani elemeket, hanem csak jelöléseket határozott meg. Ezért a '96-ban összeállított következő, 0.9-es változatát átnevezték UML-re (Egységesített modellező nyelv), melyhez a Jacobson-féle Objectory-t ajánlották módszertannak. Ugyanakkor hangsúlyozták, hogy az UML bármely más modellező eljárással is használható.

Időközben a legtöbb komoly informatikai cég is felsorakozott az Egységesített modellező nyelv mögé. A jelölésrendszert a `UML Partners`-nek nevezett csoport véleményének és ötleteinek figyelembevételével módosították és együttesen indították el az UML szabványosítását is.

Mivel az UML-t a három legelterjedtebb módszer és jelölési rendszer megalkotója állította össze, ezért az már önmagában „de facto" szabvánnyá vált volna. A szerzők azonban kezdettől fogva törekedtek arra, hogy a jelöléseket jogilag is szabványként fogadják el.

szabványosítás

Az UML 1.0-s változata '97 januárjában született meg, amelyet a szerzők már szabványosításra készneknek és ipari méretekben is alkalmazhatónak ítéltek. Ezzel a változattal kapcsolatban azonban több kifogás is felmerült, elsősorban a metamodell oldaláról. A szeptemberre elkészült 1.1 változatban már kijavították a következetlenségeket és egységesítették a jelöléseket, mely változat novemberre hivatalosan is szabvánnyá vált.

Az UML dokumentációit a Rational web-lapjain keresztül bárki elérhette. Ezek a dokumentumok azonban alapvetően technikai nyelvezetet használtak, így a jelölésrendszerrel való ismerkedésre gyakorlatilag alkalmatlanok voltak. Érdemes ezért megemlítenünk Fowler és Scott '97-ben megjelent *UML Distilled. Applying the Standard Object Modeling Language* című könyvét, amely közérthető módon mutatta be az UML-t és arra is adott javaslatokat, hogy a szoftverfejlesztés fázisaiban hogyan használjuk fel az egyes jelöléseket.

Fowler és Scott

A következő évben a „három amigo" három könyvet írt. Mindegyik könyvön mindhárman szerzőként jelennek meg, csak éppen a sorrend más:

- A Booch, Rumbaugh és Jacobson: *The Unified Modeling Language User Guide* az UML jelöléseinek és tipikus felhasználásának bemutatását tűzte ki célul.
- A Rumbaugh, Jacobson és Booch: *The Unified Modeling Language Reference Manual* részletesen tárgyalja az UML elemeit.
- A Jacobson, Booch és Rumbaugh: *The Unified Software Development Process* a Jacobson-féle Objectory, a Booch módszer és a Rumbaugh vezetésével összeállított OMT elemeiből építkező Egységesített szoftverfejlesztési eljárást tárgyalja.

Időközben a szabványosítást irányító szervezet, az OMG *(Object Management Group)* kéréseinek, illetve közreműködésének segítségével '98 júniusában megszületett az UML 1.2-es, ősszel pedig az 1.3-as változata, mely utóbbit tárgyalja a három könyv. A szerzők '99 közepére ígérik az 1.3-as változat végleges dokumentációját.

UML 1.2 és 1.3

Mi az UML?

Az OO rendszerfejlesztési módszerek történetének ismeretében összefoglalhatjuk az UML jellemzőit.

Az UML

- a Unified Modeling Language, magyarul *Egységesített modellező nyelv* elnevezés rövidítése.
- az *objektumorientált* szemléletre épülő *elemzés és tervezés eszköze*. Az UML kiterjedt eszközkészlettel rendelkezik az OO koncepciók jelölésére, bár használható strukturált módszerrel elkészített modellek ábrázolására is.

OO elemzés és tervezés eszköze

- közvetlen *örököse* a korábbi három legnépszerűbb objektumorientált módszertan, a *Booch módszer*, a Rumbaugh és társai által összeállított *OMT*, valamint a Jacobson-féle *Objectory/OOSE* jelöléseinek. A folytonosságot maguk a módszertanok szerzői, illetve vezetői biztosítják, akik együtt fejlesztik az UML-t, amely így a három jelölésrendszer következő generációjának is tekinthető.

Booch '93, OMT és Objectory örököse

- jogilag is *szabványos* jelölésrendszer.

szabványos

- alapvetően *grafikus* nyelv, azaz a modellt különböző diagramok segítségével ábrázolja.
- *modellező nyelv*. A „modellező" jelző arra utal, hogy az UML nem vizuális programozási nyelv, ezért segítségével csak az alkalmazás vázlata ábrázolható, a teljes implementáció azonban nem. Az UML „nyelv" jellege pedig azt jelenti, hogy az UML a modellek ábrázolására alkalmas jelölésrendszer és nem módszertan (nem ad módszert a fejlesztés lépéseire).

grafikus modellező nyelv

szabványos nyelv jelentősége

A szabványosságnak elsősorban az információcserénél van jelentősége, ezért ez a kulcs az OO módszerek ipari méretekben történő alkalmazásához. A „közös nyelv" egy fejlesztői csoporton belül is segíti a kommunikációt, de az UML alkalmazható a vállalatok, esetenként a megrendelő és a fejlesztő cég közötti információcsere eszközeként is. Például elképzelhető, hogy egy tanácsadó vállalat az elemzési szintű modellt készíti el, amelyet egy másik vállalat tervez meg, majd végül egy harmadik implementál.

Ha egy fejlesztői csoport az UML-t választja, akkor nem szükséges külön belső szabványt kialakítani, hanem az készen felhasználható. Egy szabvány gyakorlati alkalmazása esetén „nem vagyunk egyedül", mivel nagyobb ismeretanyagra, könyvekre, cikkekre vagy éppen szakértőkre támaszkodhatunk. Mivel az UML-t főiskolákon és egyetemeken is oktatják, ezért lerövidíthető az újonnan felvett képzett informatikusok munkába állása.

Nyelv és módszer

Az UML önmagában csak a modellek ábrázolására alkalmas, a fejlesztéshez azonban egy módszer is szükséges. Az UML-t használhatjuk az OMT módszertani elemeivel, mely tiszta és világos fogalmakkal dolgozik, eljárása azonban a napjainkra összegyűlt tapasztalatok alapján kissé naivnak tűnik. A Rational vállalat az Objectory utódját, a Rational Unified Process-t (RUP) ajánlja, mely már több modern elemet is alkalmaz. Sajnos a RUP az alkalmazásfejlesztést csak nagyon vázlatosan tárgyalja, így a modellezés és megvalósítás mindennapi problémáira alig találunk megoldási javaslatokat.

Az objektumorientált szemléletre épülő rendszerszervezési módszerek alapvetően két eszközre épülnek.

A *modellező nyelv* alkalmas a vázlatos tervek, a modellek ábrázolására. Az OO módszerek alapvetően grafikus nyelvet, azaz *diagramokat* használnak. A nyelv ismerete egy kész modell értelmezését teszi lehetővé, de nem ad ajánlást arra vonatkozóan, hogy azt milyen lépésekkel állítsuk elő.

modellező nyelv

A *módszer* (más néven *procesz* (`process`) vagy *módszertan*) felajánl egy olyan lépéssorozatot, amely segítségével megrajzolható az alkalmazás modellje.

modellező módszer

A modellező nyelv ismerete önmagában csak arra elegendő, hogy egy elkészített modellt értelmezzünk. Ennél több nem is szükséges például a programozók („kódolók") esetén, akik a modell alapján kidolgozzák a számítógépes megvalósítást. Ha egy műszaki végzettségű megrendelő ellenőrizni szeretné az elkészített terveket, akkor számára is elegendő csak a modellező nyelv jelöléseit elsajátítani.

A rendszerszervezés elemzés és tervezés fázisaihoz azonban egy alkalmazásfejlesztési módszer ismerete is szükséges, amely megadja egy tipikusnak tekintett fejlesztés lépéssorozatát, amit vezérfonalként használhatunk.

Fontos hangsúlyoznunk, hogy az UML mindössze egy modellező nyelv. A „három amigo" szándékosan választotta szét a nyelvet és a módszert. Az UML így egy olyan jelölési rendszer, amely elvileg bármely rendszerfejlesztési módszerrel használható.

az UML egy modellező nyelv

Az OMT jellemzői

Az UML bizonyos szempontból az OMT nyelvének bővítéseként is tekinthető, így a rendszerfejlesztés során egy kézenfekvő lehetőség az OMT módszertani elemeire támaszkodnunk.

Az OMT mind nyelvi, mind módszertani szempontból több előnnyel is rendelkezik. Modellező nyelvének jelölései tiszta és világos fogalmakra épülnek, így az OMT könyve az OO szemlélettel történő ismerkedésként is felhasználható. Az OMT

egyszerű és könnyen érthető módszer

módszertana az Abbot által javasolt egyszerű módszeren, a probléma szöveges megfogalmazásának nyelvtani elemzésén („húzzuk alá a főneveket", stb.) alapszik. A fejlesztésre konkrét és gyakorlatias tanácsokat ad, ezért ugyancsak alkalmazható az OO fejlesztés bevezetésére és tanulására.

Publikálásának idején az OMT nyelvét „rémisztően részletesnek" találták, a gyakorlati problémák esetén a jelölések mégis kevésnek bizonyultak. Az UML már sokkal több eszközzel rendelkezik, bonyolult jelölésrendszere azonban megnehezíti a megértését és a megtanulását. A bonyolultság egyik oka hogy több finomítás ellenére a nyelv még mindig sok inkonzisztenciát tartalmaz. Másik hátránya, hogy az UML a jelölések mögött meghúzódó fogalmakat, a „szemléletet" már nem írja le olyan tisztán és közérthetően, mint azt az OMT tette.

Az OMT módszere egy jól, szinte gépiesen követhető lépéssorozatot határoz meg, ezért első pillantásra meglepő, hogy azt a konkrét rendszerfejlesztések során szinte alig alkalmazták. A publikálásakor még forradalmi módszer a gyakorlatban naivnak és kevésnek bizonyult. Az OMT legtöbb lépése annyira egyszerű, hogy azok eredményei kis gyakorlattal minden különösebb eljárás követése nélkül azonnal előállíthatók, így ezek a lépések inkább lassítják, mint segítik a modellezést. Ugyanakkor a módszer a rendszerszervezés több gyakori problémájára semmilyen választ sem ad.

a módszer ma már kevés tanácsot ad

A korai objektumorientált módszertanok általános hiányossága volt, hogy nem adtak segítséget a kifejlesztendő alkalmazással szemben támasztott követelmények tisztázására. Az OMT eredeti változata is a feladat szövegesen megfogalmazott leírásából indul ki, bár a második változat egy bevezető lépésként már a használati esetek összegyűjtését ajánlotta. Ez azonban sajnos csak egy utólagos hozzátoldás és nem illeszkedik szervesen a módszerbe.

Rational Unified Process

Amikor Jacobson, illetve az Objectory nevű vállalata csatlakozott a Rational-nél már együtt dolgozó Booch-hoz és Rumbaugh-hoz, akkor a közös munka során felvetődött több kérdés is egyszerre megoldódott. Az UML előde, az első év eredményeként megszülető Unified Method (Egységesített módszer) a neve ellenére „csupán" egy modellező nyelv volt és semmilyen módszertani elemet sem tartalmazott. A Rational az Objectory-val együtt az annak tulajdonában lévő Objectory módszertant is átvette. Ezután választották ketté az átnevezett Egységesített modellező nyelvet (az UML-t) és a módszertant (az Objectory-t).

Amíg az UML-t a szabványosításra alkalmassá tették, addig a Rational-nél az Objectory-t is folyamatosan fejlesztették, így a Booch módszer, az OMT, valamint néhány más módszer elemeivel is kiegészítették. Egy nagyobb átdolgozást követően, 1998 végén az Objectory-t Rational Unified Process-re („Rational-féle egységesített eljárás", rövidítve: RUP) nevezték át.

Objectory örököse

A Rational, illetve a „három amigo" által az UML-hez és általában az objektumorientált rendszerfejlesztéshez ajánlott Rational Unified Process már nagyon sok modern elemet tartalmaz.

iteratív és inkrementális

A RUP a rendszerfejlesztésre iteratív és inkrementális eljárást ajánl. Az *iteratív* jelző azt jelenti, hogy a teljes alkalmazást nem „vízesésszerűen" egyszerre, a teljes körű elemzés, majd a teljes körű tervezés és implementáció lépéseit követve valósítjuk meg. Ehelyett az igényeket több részre bontjuk és először csak a leglényegesebb elemeket valósítjuk meg, majd a következő iterációs ciklusban újabb részeket fejlesztünk ki. Minden ciklus végén egyre több ismeret gyűlik össze az alkalmazással szemben támasztott tényleges követelményekről és a megvalósítás lehetőségeiről, így jelentős mértékben csökkenthetjük a fejlesztés kockázatait. Az előzetes iterációk kivételével minden iteráció működő és tesztelhető szoftvert állít elő. Az *inkrementális* jelző arra utal, hogy az egyes iterációs lépések során nem egymástól független elemeket valósítunk meg, hanem mindig valami többletet, valami inkrementumot teszünk hozzá a készülő rendszerhez. Az elemek integrációja így folyamatos, nem pedig egy utolsó nagy és éppen ezért bizonytalan kimenetelű lépésben történik. A két elv eredetét Booch módszerének az „elemezzünk egy keveset, tervezzünk egy keveset, majd implementáljunk egy keveset" technikájában találhatjuk meg.

használati eset vezérelt

A RUP módszere az Objectory örökőseként a használati esetek segítségével irányítja a fejlesztés teljes menetét a követelmények tisztázásától egészen a végső tesztelésig és integrációig. A használati esetek meghatározása segíti például az egyes iterációs lépések során megvalósítandó elemek kiválasztását.

ellenőrzött

A használati esetek és az iterációs ciklusok miatt a fejlesztés folyamata ellenőrzött, azaz biztosíthatjuk, hogy valóban a felhasználók által igényelt alkalmazást fejlesztjük ki a kívánt minőségben. Mivel nagyobb rálátásunk van a követelményekre, ezért a fejlesztés esetleges kockázatai, erőforrásigénye és költségei is jobban megbecsülhetők, valamint magát a fejlesztés folyamatát is megtervezhetjük. Az ellenőrzés a vállalaton belüli minőségbiztosítást is lehetővé teszi, mely nem csak egy független tevékenységként hajtható végre, hanem az a fejlesztés menetébe integrálható.

architektúra központú modellezés

Az Egységesített eljárás már a fejlesztés korai szakaszában hangsúlyozza az architekturális tervezés fontosságát, amelyet körülbelül úgy fordíthatunk le, hogy az alkalmazást a jól kialakított és összehangoltan működő komponensekből építjük fel. A robosztus architektúrára összpontosító tervezés lehetővé teszi a párhuzamos fejlesztést, az újrafelhasználhatóságot és egyszerűsíti a későbbi karbantartást.

a fejlesztéssel kapcsolatban kevés segítség

Sajnos, a Rational Unified Process a fejlesztést csak vázlataiban tárgyalja, a konkrét modellezésre és megvalósításra azonban alig találunk segítséget. Ez a vázlat alkalmas a fejlesztés főbb menetének, fontosabb mérföldköveinek a nyomon követésére, így segítségével a fejlesztésért felelős és azt irányító menedzserek végigkísérhetik a folyamatot és számon kérhetik az eredményeket.

Önmagában a RUP a már (OO területen is) nagy gyakorlattal rendelkező fejlesztők kézikönyveként használható, akik számára csak a fő irányt kell kijelölni, mert a fejlesztés mindennapos feladatai nem jelentenek problémát. Ugyancsak hátrány, hogy az objektumorientált szemlélet „hagyományát" örökölve az Egységesített eljárásban nem válik élesen szét az elemzés és a tervezés, ezért az elemzés könnyen tervezésbe „csúszhat" át.

A „Q" modellező módszer

*A könyv célja az UML megismertetése és a **Q** modellező módszer bemutatása. A **Q** módszer a fejlesztés folyamatát helyezi előtérbe. Célja, hogy a fejlesztőket egyetlen pillanatra se hagyja bizonytalanságban, hanem mindig ajánlja fel a következő lépést, valamint mutassa meg, hogy a munkának mikor lesz vége. A módszer segítségével tipikus alkalmazás esetén megtervezhetjük és áttekinthetjük a fejlesztés menetét.*

A könyv célja az Olvasót egy könnyen érthető és követhető objektumorientált alkalmazásfejlesztési módszer birtokába juttatni, amely szabványos jelölésrendszerrel ábrázolja az alkalmazással kapcsolatos vázlatos terveket, a fejlesztés „közbülső" eredményeit. A „közbülső" jelző azért került idézőjelbe, mert a közismert mondás szerint egy alkalmazás fejlesztése csak akkor fejeződik be, ha azt az összes létező adathordozóról letöröljük, beleértve a kinyomtatott dokumentumok megsemmisítését is.

A következők tehát egyrészt a szabványos jelölésrendszert, azaz az UML-t ismertetik, másrészt pedig egy, a szerző által kidolgozott eljárást, amely a „***Q***" elnevezést kapta. Azért, hogy az UML jelölései érthetőbbek és világosabbak legyenek, a modellezési nyelv ismertetését összekapcsoljuk az alkalmazásfejlesztési módszer egyes elemeinek a bemutatásával.

A „***Q***" elnevezés részben a minőségre és a gyorsaságra utal (`Quick modeling method`: gyors modellező módszer), mivel egy gyors, mégis áttekinthető, követhető és ellenőrizhető fejlesztési stratégiát tesz lehetővé. A módszer az OMT, a Rational Unified Process, valamint más módszerek elemeiből is építkezik, azonban jelentős kiegészítésekkel áthidalja az OMT régiességéből és naivitásából, valamint a RUP vázlatosságából és menedzseri szemléletéből fakadó nehézségeket.

a fejlesztés folyamatára helyezi a hangsúlyt

A módszer legfontosabb jellegzetessége, hogy a *fejlesztés folyamatát helyezi előtérbe* és viszonylag kevés figyelmet fordít egyrészt a kiegészítő, például menedzseri tevékenységekre, másrészt a fejlesztés részleteire, például arra, hogy milyen módon oldjuk meg a csoportmunka során a fájlok közös használatát, hogyan valósítsuk meg a megrendelővel történő találkozókat vagy a felhasználói felületeken milyen elrendezést kövessünk. Ezért elsősorban kisebb (kb. 25 fő alatti) dinamikus fejlesztőcsoportok, illetve maguk a fejlesztők számára hasznos.

egyszerűen követhető lépéssorozat

A ***Q*** modellező módszer olyan fejlesztői csoportoknak adott tanácsokból fejlődött ki, amelyek gyakran nem rendelkeztek objektumorientált fejlesztési tapasztalattal, sőt, esetenként néhány hónapos munkaviszonyú, frissen végzett programozókból álltak. A tapasztalatot olyan lépéssorozattal kellett helyettesíteni, amelyet szinte automatikusan végigkövetve a fejlesztők elő tudták állítani a kész alkalmazást.

A ***Q*** módszer célja, hogy a fejlesztőket egyetlen pillanatra se hagyja bizonytalanságban, hanem mindig ajánlja fel a következő lépést, valamint mutassa meg, hogy a munkának mikor lesz vége. Ezért egy olyan „gépnek" is tekinthető, melybe a követelményeket és az alkotók kreatív fantáziáját bedobva félautomatikusan kitermeli az alkalmazást. A jól definiált lépéssorozat lehetővé teszi, hogy figyelmünket és kreativitásunkat ne a fejlesztés következő lépésének a kitalálása kösse le, hanem intelligensebben működő minőségi alkalmazásokat teremthessünk. A ***Q*** módszer így nem a részletekre, hanem a fejlesztés *stratégiai és taktikai* lépéseire összpontosít.

a fejlesztés stratégiai és taktikai lépései

A módszer *ellenőrzött* fejlesztést tesz lehetővé, azaz segítségével biztosíthatjuk, hogy valóban a megrendelők által igényelt rendszert fejlesszük ki, indokolható költségű munkával. A legvázlatosabb modellek segítségével a felhasználó beleláthat a tervekbe, annak ütemezésébe és kritikus pontjaiba.

ellenőrzött fejlesztés

A ***Q*** modellező módszert követve *„piramis"-jellegű* módon építkezünk. Először az alkalmazás absztrakt modelljét készítjük el, amelyet a későbbiekben fokozatos „leszállással" lépésenként pontosítunk. A modellek megrajzolásával biztosítjuk, hogy az alkalmazással kapcsolatos terveket mindig át tudjuk tekinteni. A piramis minden egyes lépcsője egy minőségileg más absztrakciós szintet jelent. A ***Q*** módszer egy előnyös tulajdonsága, hogy a legtöbb objektumorientált módszertől eltérően az egyes szintek élesen elválnak egymástól, bár a modellek ábrázolására azonos jelölési rendszert alkalmazunk. A piramis talpazata a követelményeknek megfelelően működő, megvalósított alkalmazás.

„piramis"-jellegű építkezés

A fejlesztés nem „vízesés-jellegű" módon, azaz a teljes rendszer elemzésével, majd a teljes tervezéssel és végül a megvalósítással történik. A ***Q*** módszer a vázlatok pontosítására a „`centre-out`" („középről kifelé") elvet ajánlja, azaz először a hangsúlyos, majd ezt követően a normál és végül a kiegészítő elemekre fordítjuk a figyelmünket. A kidolgozás során mindig felveszünk egy lényegesnek ítélt szálat, amit lefuttatunk a pontosítás néhány lépésén, esetenként egészen az absztrakció következő lépcsőfokáig. Ezután kiválasztjuk a következő lényeges szálat és azon is végrehajtunk néhány lépést.

„centre-out"

A ***Q*** módszer is *iteratív és inkrementális*. Az iterációs ciklusok a fejlesztés nagyobb léptékű beosztását teszik lehetővé és a bevezető iterációk kivételével mind működő, ezért tesztelhető és a megrendelőknek bemutatható alkalmazást valósítanak meg. Az egyes iterációk feladatát is a „`centre-out`" módszerrel határozzuk meg.

iteratív és inkrementális

A fejlesztés lépcsőfokai

A ***Q*** modellező módszer esetén az iteratív fejlesztés egyes fázisai (elemzés, tervezés és megvalósítás) élesen szétválnak, ezért egyrészt ismert, hogy még mennyi munka szükséges ahhoz, hogy teljesítsük a fázis által előírt célt, másrészt minden dokumentumról megállapítható, hogy az mely fázisra vonatkozik.

Elemzés

A fejlesztést a feladat *elemzésével* kezdjük, mely célja az alkalmazásnak kizárólag a szakterület fogalmaival történő pontos leírása. Az elemzés során a *koncepciókra* összpontosítunk és kerüljük a megvalósítással kapcsolatos technikai részletek és informatikai fogalmak használatát. Az elemzés két fázisra bontható:

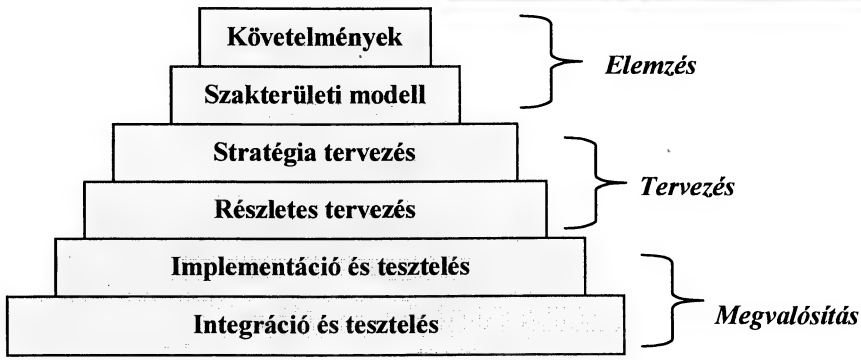
követelményelemzés

- A *követelményelemzés* során az alkalmazásnak a felhasználók által látható külső képét vázoljuk, melyet a megrendelővel való folytonos egyeztetések során állítunk elő. A követelményelemzés a felhasználó által minden részletében megérthető dokumentumot állít elő.

szakterületi elemzés

- A *szakterületi elemzés* során az alkalmazás fogalmait, a közöttük lévő viszonyokat, illetve a részek együttműködéseit és változásait rögzítjük. Az előző fázisban tisztázott követelmények, alapkoncepciók közvetlenül nem fogalmazhatók át az alkalmazássá, mivel csak közvetve utalnak a felépítendő belső szerkezetre, az architektúrára. A szakterületi modell az első mérnöki-jellegű produktum, ami a belső összefüggéseket tisztázva már nem az igények, hanem a fejlesztés szempontjából közelíti meg a problémát.

11. Ábra: A fejlesztés lépcsőfokai



Tervezés

Az elemzést követő *tervezés* a szakterületi modellt az informatika világába ülteti át. Célja az alkalmazásnak az informatika fogalmaival történő pontos leírása, mely alapján a kiválasztott programozási nyelvekhez, környezetekhez és technológiákhoz (pl. grafika, multimédia vagy éppen statisztika) értő programozók önállóan meg tudják valósítani a feladatot. Ez a technikai megközelítés ugyancsak két fő fázisra bontható:

alkalmazási környezet

- A *stratégiai tervezés* feladata a megvalósítással kapcsolatos nagyobb léptékű döntések meghozatala. Ekkor határozzuk meg például a befogadó környezetet, a programozási nyelveket és az adatbázis-kezelőt is.

részletes tervezés

- A *részletes tervezés* elsősorban a modell részeire összpontosítva informatikai szempontból is teljes mértékben megadja az alkalmazást. Ebben a lépésben jelöljük ki az infrastruktúrát, amely a szakterületi modellben meghatározott rendszert működtető kiegészítéseket, kifejlesztendő osztálykönyvtárakat és keretrendszereket jelent.

A tervezést követő *megvalósítás* az elemzés során vázolt alkalmazás megvalósítását jelenti a tervezés során meghatározott technológiák felhasználásával. A megvalósítás szintén két fázisra bontható. *Megvalósítás*

- Az *implementáció és teszt* az alkalmazás részleteinek megvalósítása adott számítógépes környezetben (operációs rendszerben, nyelven, stb.), mely összekapcsolódik a megvalósított elemek ellenőrzésével, hogy működésük megfelel-e az elemzés és a tervezés során rögzített követelményeknek. *implementáció és teszt*
- Az *integráció és együttes teszt* a különállóan implementált elemek beillesztését jelenti az alkalmazás egészébe, mely összekapcsolódik az újonnan kifejlesztett elemek és a teljes alkalmazás együttműködésének vizsgálatával. *integráció és együttes teszt*

Kiegészítő ajánlások és értelmezések

Az UML-szabvány csak nagyon durva körvonalakban definiálja az egyes jelölések jelentését, így tág teret ad a modellek ábrázolására és a különböző értelmezésekre. Ez egyrészt előnyös, mivel a modellező nyelv a feladatok széles tartománya esetén alkalmazható és a használók kezét sem köti meg. Ugyanakkor ez azt is jelenti, hogy az UML-t nem ajánlatos önmagában használni. Célszerű összeállítani egy kiegészítő belső szabványt, mely a jelölések elnevezésével és elhelyezésével kapcsolatos ajánlásokat vagy éppen kötelező érvényű szabályokat tartalmazza, valamint a *leképezés* (`mapping`) módját, mely megadja, hogy a jelölések pontosan mit is jelentenek. Ezek hiányában az ábrázolások kaotikusakká válhatnak, a diagramokat pedig többnyire csak szemléltetésre használhatjuk fel, mivel a benne foglalt információkat nem tudjuk automatikusan kóddá transzformálni. *szabályok* *leképezés*

A ***Q*** modellező módszer a modellek kialakítására a jelenlegi általános gyakorlatnál jóval több ajánlást ad, a jelölésrendszernek majdnem az összes elemére pedig egzakt(!) leképezést határoz meg, mely a jelöléseknek objektumorientált informatikai technikákat feleltet meg.

Az egzaktság miatt az elkészített modell egy automatikus eljárással kóddá alakítható, illetve a diagramokat értelmezve pontosan tudjuk, hogy az egyes elemek miként jelennek meg a kódban (a reprezentációban). Meg kell azonban jegyeznünk, hogy az UML jelöléseivel a teljes számítógépes reprezentáció nem ábrázolható minden részletében, így általában a programutasítások sorozata sem.

A szemlélet

A ***Q*** modellező módszer egy érdekessége, hogy nem tisztán az objektumorientált szemléletre épül, hanem az OO technikákat olyan módon használja fel, ahogy az OO a strukturált megközelítés eredményeit. Az objektumorientáltság az objektumokat, illetve azok belső felépítését tekinti lényegesnek, képletesen: az alkalmazást mintegy objektumokra robbantja szét. Ezen új szemlélet nem az objektum belső tartalmára helyezi a hangsúlyt, hanem ezzel ellentétben annak környezetével vett kapcsolatára,

információs csomópont

más entitásokkal való viszonyára. Az egyes objektumokat/entitásokat a rendszer információs csomópontjaiként értelmezi. A két megközelítés között körülbelül az a különbség, mint egy független objektum és egy komponens-architektúra komponense között.

használat, kapcsolódási mód

A szemlélet alapja a modellelemek és viszonyaik jellegének megragadása és kiemelése, mely jelleg elsősorban a környezettel való kapcsolódást, azaz a használatot, az elemmel szemben támasztott követelményeket jelenti. A „használat" előtérbe helyezése miatt a módszer őseiként tekinthetők a felelősségeken alapuló tervezés *(RDD: Responsibility Driven Design)*, a komponens-architektúrák, az elemzési és tervezési minták, a szempont-orientált programozás *(AOP: Aspect Oriented Programming)*, stb. A szemlélet ezeket az átfogóbb vagy éppen a részleteket érintő eredményeket egyetlen gondolatkörbe fogja össze.

A ***Q*** modellező módszer a szemlélet következetes alkalmazása érdekében esetenként eltér az UML jelöléseinek szabványos értelmezésétől, ezekre az eltérésekre azonban mindig felhívjuk a figyelmet. A módszer olyan jelöléseket is megenged, amelyek még nem kerültek be az UML-be, de annak jelölései között hasonló értelmezéssel már megtalálhatók. Így valószínű, hogy valamely későbbi változatban ezek is teljes jogú szabványos jelölésekké válnak.

Hogyan használjuk?

Természetesen dönthetünk úgy, hogy a ***Q*** módszert egyáltalán nem alkalmazzuk, hanem hűségesek maradunk a régi bevált rendszerfejlesztési eljárásunkhoz, melyet esetleg az UML követelményeinek megfelelően módosítunk. A ***Q*** módszer bizonyos elemeit ekkor is felhasználhatjuk a konkrét fejlesztés során felmerülő egyes részproblémák megoldásakor.

fokozatos bevezetés

A módszer túlságosan sok újdonsága miatt egy vállalatnál az azonnali és teljes körű bevezetése valószínűleg nem hozna gyors eredményeket. Ehelyett kezdésként célszerűbb a technikákból összeválogatni egy áttekinthető és könnyen értelmezhető „túlélési készletet" és azt kipróbálni a gyakorlatban. Több jelölésnek is van egy „szelídített", a szabványhoz és a hagyományos megközelítésekhez közelebb álló változata. Ha az induló készlet technikái bizonyítottak, akkor azokat fokozatosan bővíthetjük a feladataink által igényelt újabb elemekkel.

A „*Q*" modellező módszer és a RUP

A Rational Unified Process sajnos csak vázlatosan írja le a fejlesztés menetét. Ez a vázlat azonban alkalmas arra, hogy a fejlesztésért felelős és azt irányító menedzserek nyomon követhetik a folyamatot és számon kérhetik az eredményeket. A ***Q*** modellező módszer a fejlesztésre helyezi a hangsúlyt, azonban az azt kísérő folyamatokat — így a menedzseri tevékenységeket is — figyelmen kívül hagyja. Az elnevezése részben ezért módszer *(method)* és nem módszertan. (A másik ok történeti, az angol OO módszerek (például: a Booch módszer) elnevezését követi.)

A Rational Unified Process fejlesztéssel kapcsolatos tanácsait az un. javaslatok *(guideline)* tartalmazzák. Fokozatosan bővülő számuk ellenére azonban ezek a fejlesztésnek még mindig csak kevés részletét fedik le.

Mivel az alkalmazásfejlesztés különböző területeit célozzák meg, a Rational Unified Process és a ***Q*** módszer jól kiegészítik egymást. A RUP alkalmazásával a fejlesztési technológiánkat is közel szabványos keretek köré szervezhetjük. A ***Q*** módszer elemeit pedig a fejlesztés mindennapi problémáiban, az éppen aktuális feladatok megoldásában alkalmazhatjuk. A RUP szempontjából a ***Q*** módszert egyfajta konzisztens javaslat-együttesként tekinthetjük.

I.

Követelmények rögzítése

A fejlesztés első lépése a követelményelemzés, amely a készítendő alkalmazás felhasználó által látható képét írja le, azaz rögzíti az alkalmazással szemben támasztott felhasználói követelményeket.

A követelmények összegyűjtése során elkészített dokumentumok a felhasználó fogalmaival és alapvetően szöveges formában vázolják az alkalmazást, így alkalmasak a megrendelővel történő egyeztetésre. Az egyeztetések segítségével biztosítható, hogy valóban az igényelt rendszert fejlesszük ki. A követelményelemzés legfontosabb célja, hogy megrajzoljuk az alkalmazás határait és alapvető részeit. A határok megadják, hogy mi kerüljön be és mi maradjon ki az alkalmazásból, az alapvető részek meghatározásával pedig megbecsülhető a fejlesztés erőforrásigénye és költsége.

Megjegyzések, fogalomszótár

UML diagramokon tájékoztató jellegű megjegyzést vagy a fejlesztéssel kapcsolatos kiegészítő információt a „noteszlap" alakzaton helyezhetünk el. Más modellelemen belül szöveges megjegyzést a -- jel után írhatunk. A modellek készítése során általában rövidített elnevezéseket alkalmazunk, ezért célszerű minden elnevezést egyszer bekezdésben kifejteni, hogy azt milyen értelemben használjuk a rendszerünk világában. Az összegyűjtött elnevezések és értelmezéseik együttesen a fogalomszótárt alkotják.

Megjegyzések

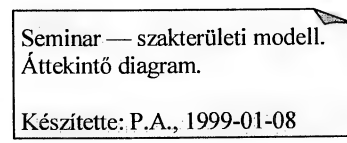
A programozási nyelvekhez hasonlóan az UML is megengedi, hogy az adott jelentéssel rendelkező formális jelölések mellett informális, azaz tájékoztató jellegű elemekkel, megjegyzésekkel tegyük a modellünket érthetőbbé.

jelölés

UML diagramokon önálló megjegyzést a „megjegyzéslap" alakzaton („noteszlapon") helyezhetünk el, melyen szöveg és kép is szerepelhet.

A noteszlapon a fejlesztéssel kapcsolatos kiegészítő információkat is rögzíthetünk.

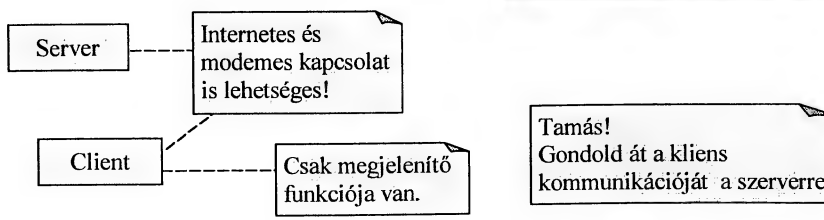
12. Ábra: Megjegyzéslap



megjegyzés csatolása

A modell adott eleméhez a rá vonatkozó megjegyzéslapot szaggatott vonallal (több elem esetén vonalakkal) csatoljuk.

13. Ábra: Megjegyzés csatolása

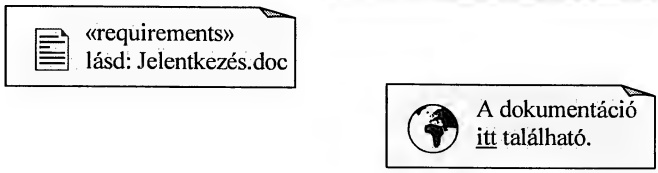


A megjegyzéseket felragasztós noteszlapokként `(stick-note)` is elképzelhetjük, melyekkel a fejlesztés aktuális feladataira vagy problémáira emlékeztető jegyzeteket vehetünk fel.

Nagyobb méretű megjegyzéseket a diagramon nem célszerű teljes egészében ábrázolni. Helyette az UML a külső dokumentumra, vagy egy Web-címre történő hivatkozást (URL-t) ajánlja.

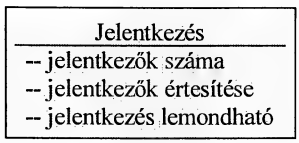
külső dokumentum

14. Ábra: Külső dokumentum és URL



Más modellelem szövegén belül a sornak a -- szimbólum utáni részét megjegyzésnek tekintjük.

15. Ábra: Szöveges megjegyzés



Elnevezések

A modellünkben felvehető elnevezéseket nagymértékben behatárolják a használt programozási nyelv és környezet szintaktikai szabályai.

Az elnevezésekben általában a latin abc betűi, számok, illetve az aláhúzásjel szerepelhet. Egyes nyelvek, például a Java az azonosítókban a nemzeti karakterek használatát is megengedi.

A modellünkben használt elnevezéseket egy egyszerű automatizmussal a kiválasztott platform szabályainak megfelelővé alakíthatjuk. Például a nemzeti karaktereket a megfelelő latin betűkkel helyettesíthetjük, ha a környezet nem engedi a nemzeti karakterek használatát (`Jelentkezés` → `Jelentkezes`).

nemzeti karakterek

Több szavas elnevezésekben a szóközöket régebben aláhúzásjelre cserélték (pl.: `jelentkezők_száma`). Ma többnyire azt a módszert alkalmazzák, hogy minden szó első betűjét nagybetűsítik, a többit pedig kisbetűsen hagyják (pl.: `jelentkezőkSzáma`, `setLocation`, stb.).

többszavas elnevezés

Fogalomszótár

Az alkalmazás fejlesztése során rövidített elnevezéseket használunk, amelyek rendszerünk zárt világában általában jól azonosítják az egyes elemeket, szereplőket. Az alkalmazás világán kívül azonban az egyes elnevezések bővebb, vagy éppen a köznapi értelmezéstől eltérő fogalmakra utalhatnak. Ezért minden egyes elnevezés esetén célszerű röviden, egy-két sorban vagy bekezdésben összefoglalni, hogy azt milyen értelemben használjuk az alkalmazás környezetében.

modell-elemek értelmezése

A modell egyes elemeinek rövid leírását az elem *értelmezésének* vagy *áttekintésének* nevezzük.

UML diagramokon az elem értelmezését egy megjegyzéslapon adhatjuk meg, amelyet az elemhez csatolhatunk.

fogalom-szótár

Az összegyűjtött elnevezéseket és az azokhoz tartozó értelmezéseket együttesen *fogalomszótárnak* nevezzük, de elterjedt még a *szójegyzék* `(glossary)` elnevezés is.

A diagramokat nem célszerű telezsúfolni az elemek értelmezését leíró megjegyzéslapokkal. Ehelyett minden egyes diagramhoz mellékeljük az azon szereplő modellelemek pontokban felsorolt értelmezéseit, azaz a fogalomszótár hivatkozott részletét.

Áttekintés

A fejlesztés kiindulópontjaként írjunk az alkalmazásról egy rövid áttekintést, melyben összefoglaljuk az alkalmazás legalapvetőbb céljait, a „víziót", és vázoljuk a szituációt, a környezetet. Az áttekintés az alkalmazásról készült legvázlatosabb terv.

A ***Q*** modellező módszer ajánlása szerint az alkalmazás fejlesztését célszerű egy rövid áttekintő leírás készítésével kezdeni. Az *áttekintés* egy-két bekezdésben, szöveges formában vázolja az alkalmazás célját és szerepét.

áttekintés

Az áttekintésben összefoglalhatjuk az alkalmazás alapvető célját, a „víziót". Itt hangsúlyozhatjuk, hogy a felhasználók munkáját milyen módon kívánjuk megkönnyíteni, ezért az áttekintésben reklámokból kölcsönzött fordulatokat is használhatunk.

alkalmazás alapvető célja

Az áttekintésben a célok mellett alapvetően a helyzetre, a szituációra kell helyezni a hangsúlyt, nem pedig a funkciókra, melyeket a későbbi lépésekben rögzítünk és pontosítunk.

szituáció

Az alkalmazásra általában egy rövid megnevezéssel utalunk. Az áttekintés valójában a teljes alkalmazást jelölő megnevezés értelmezése.

Az áttekintés a diagramon is elhelyezhető egy megjegyzéslapon.

Első példánk egy olyan alkalmazás lesz, mely egy tanfolyamszervező cég számára lehetővé teszi, hogy tanfolyamait meghirdethesse az Interneten, valamint figyelemmel kísérje a jelentkezéseket. A könyv kereteire való tekintettel a példát a töredékére egyszerűsítettük.

Lássuk tehát az alkalmazás áttekintését.

Seminar— a tanfolyamszervezés támogatása.

A *Seminar* alkalmazás megkönnyíti a tanfolyamok szervezését. Segítségével Web-en meghirdetheti tanfolyamait, melyekre az érdeklődők egy űrlap kitöltésével előzetesen jelentkezhetnek.

A tanfolyamok meghirdetését az oktatási igazgató (v. helyettese) irányítja, az adminisztratív teendőket pedig a tanfolyamszervezők látják el.

A meghirdetett tanfolyamot a megtartása előtt 3-5 munkanappal a szervező véglegesítheti vagy elegendő jelentkező hiányában, esetleg egyéb különleges ok miatt lemondhatja; mindkét esetben a jelentkezőket e-mail-ben értesíteni kell. A jelentkezések szintén lemondhatók.

A megtartott/lemondott tanfolyamok adatait egy archívum gyűjti.

Kiterjesztési mechanizmusok

A kiterjesztési mechanizmusok segítségével a szabványos jelölésrendszert „testre szabhatjuk", azaz a szakterület, valamint az alkalmazott technológia és fejlesztési módszer követelményeinek megfelelően specializálhatjuk. Az UML három kiterjesztési mechanizmusa az új modellelemek felvételére alkalmas sztereotípia, a megszorítás és a kulcsszavas érték.

UML kiterjesztése

Az UML szabványos jelölései a modellek ábrázolásának mindössze egy általános keretét határozzák meg. A kiterjesztési mechanizmusok teszik lehetővé, hogy az általános jelölésrendszert a fejlesztés által megkövetelt irányba specializáljuk. A megfelelő kiegészítésekkel az UML-t alkalmassá tehetjük arra, hogy a szabvány keretein belül az alkalmazás szakterületének, az alkalmazott technológiának és a fejlesztési módszernek megfelelő jelölésekkel is rendelkezzen.

Az UML kiterjesztési mechanizmusai a következők.

- A *sztereotípia* lehetőséget ad új modellelemek jelölésére.
- *Megszorításként* vehetjük fel azokat a tulajdonságokat, amelyek az UML más jelöléseivel nem adhatók meg.
- *Kulcsszavas értékek* segítségével a modellelemek leírását speciális jellemzőkkel egészíthetjük ki.

A kiterjesztési mechanizmusokkal az UML a fejlesztés és a feladat által igényelt környezethez igazítható.

Sztereotípia

magas szintű tipizálás

A *„sztereotípia"* `(stereotype)` elnevezés Rebecca Wirfs-Brock-tól származik, aki a technikát eredetileg az osztályok magas szintű tipizálására használta. Magas szintű tipizáláson az osztály általános céljának és jellegének megadását értjük, amelyek a kódban csak közvetve („jellegként") jelennek meg, például az osztály elláthat vezérlő `(controller)` vagy megjelenítő `(view)` funkciókat. A Jacobson és társai által publikált Objectory/OOSE módszer azt javasolja, hogy a sztereotípiák jelölésével minden osztályt három csoportba soroljunk: modell (szakterületi osztály), interfész (például megjelenítő), illetve vezérlő típusokba. A Rational Unified Process is átvette ezt a csoportosítást, mindössze a modell elnevezést entitásra `(entity)`, az interfészt pedig határosztályra `(boundary)` módosította.

új modellelemek képzése

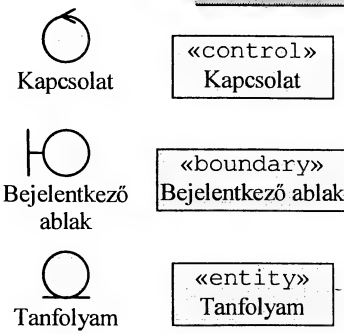
Az UML a sztereotípiákat bármely modellelem esetén megadható általános kiterjesztési mechanizmusként alkalmazza, mellyel új modellelemeket képezhetünk.

jelölés

Modellelem sztereotípiáját francia-idézőjelek (« és ») között az elem neve előtt, vagy fölött adhatjuk meg (például «`control`»).

Minden sztereotípiához kis kép („ikon") is rendelhető.

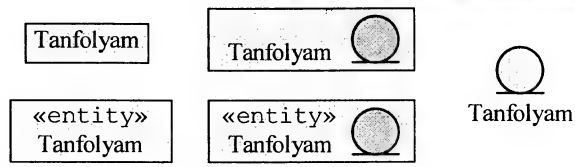
16. Ábra: A Unified Process ikonjai



A sztereotípia megadható szövegesen (francia idézőjelek között) a név előtt vagy fölött, grafikusan az elem elnevezése mellett, vagy mindkét módon. Egy modellelem vázlatos ábrázolásakor csak a sztereotípia ikonját és alatta az elem elnevezését jelenítjük meg.

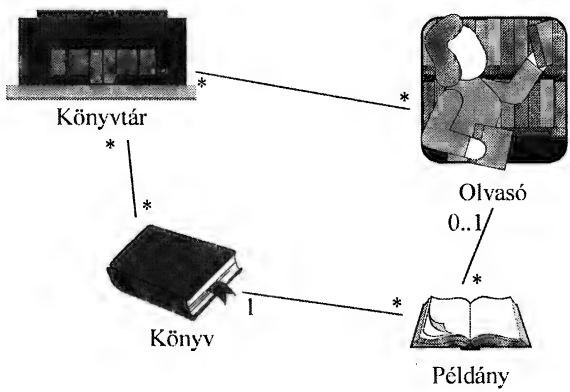
ikon

17. Ábra: Szöveges és grafikus ábrázolás



Ikonok segítségével szemléletesebbé tehetjük ábráinkat. Ez különösen a felhasználók számára készített dokumentumok esetén hasznos, melyek így egyszerre szabványosak és könnyebben is értelmezhetők.

18. Ábra: Ikonok



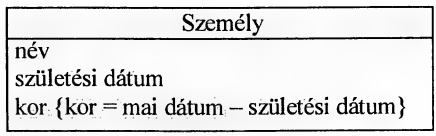
Megszorítások

A modellező nyelvek minden egyes jelölése arra szolgál, hogy segítségükkel meg tudjuk fogalmazni a modellel kapcsolatos információkat. A definiált jelölések mellett a szöveges *megszorítások* `(constraint)` olyan általános eszközként használhatók, melyekkel a modellünket tovább pontosíthatjuk. A megszorítás lehet egyetlen elemre vonatkozó tulajdonság, de leírhat több elem közötti kapcsolatot is.

jelölés

Megszorításokat az UML jelöléseivel az elem elnevezése alatt vagy mögött, kapcsos-zárójelek között adhatunk meg. A megszorítás lehet szöveges (például magyar vagy angol, stb. nyelven), vagy formális, például egy programozási nyelv kifejezése vagy matematikai képlet.

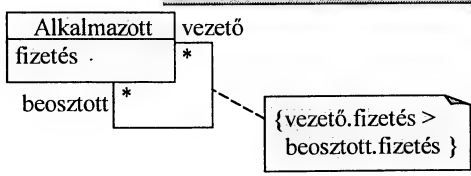
19. Ábra: Megszorítás



megjegyzésen

A megszorítás külön megjegyzéslapon is megadható, amelyet az elemhez vagy elemekhez szaggatott vonallal csatolunk.

20. Ábra: Megjegyzésként megadott megszorítás



Az UML szerzői a megszorítások formalizált megadására az eredetileg az IBM-nél kifejlesztett OCL-t `(Object Constraint Language)` ajánlják.

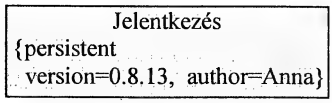
Megszorítás és kulcsszavas érték

Az egyszerű megszorításként megfogalmazott, a modellelemek által felvehető értékekre vonatkozó szabályok mellett az elemek jellegét, használati módját is megadhatjuk. Például egy `{frozen}` („befagyasztott") megszorítás jelöli, ha az elemet a kezdőértékének beállítása után már nem változtathatjuk.

A különleges megszorításokhoz hasonlóak a *kulcsszavas értékek* `(tagged value)`, melyekkel adott névhez értéket rendelhetünk. A fejlesztéssel kapcsolatos információkat is kulcsszavas értékekként rögzíthetjük.

Érték önállóan is szerepelhet, ha a jelentése egyértelmű, például az egy felsorolásos típuson belüli elem (pl.: `transient` vagy `persistent`).

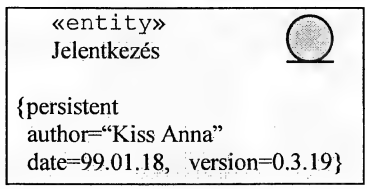
21. Ábra: Kulcsszavas érték



Kiterjesztési mechanizmusok használata

A sztereotípiák, a megszorítások és a kulcsszavas értékek egyaránt az UML kiterjesztési mechanizmusai. Segítségükkel a szabványos jelölésrendszer keretein belül a modell pontosabb ábrázolásához szükséges további jelöléseket adhatunk meg. Néha nehéz eldönteni, hogy adott esetben melyik kiterjesztési mechanizmust alkalmazzuk.

22. Ábra: Sztereotípia és megszorítás



A sztereotípiával újabb modellelemet, azaz egy metatípust (vagy más néven „koncepcionális típust") definiálhatunk. *metatípus*

Kulcsszavas értékként, illetve speciális megszorításként a modellelemeket leíró újabb tulajdonságot, azaz metadatatot (vagy más néven „koncepcionális attribútumot") határozhatunk meg. *metaadat*

Egyszerű megszorításként egy vagy több modellelemre vonatkozó összefüggést, szabályt fogalmazhatunk meg. *szabály*

A metatípus és a metaadat a modellelem olyan jellemzőire utal, melyek az alkalmazásban nem közvetlenül, hanem csak közvetve, „jellegként" jelennek meg.

Használati esetek

A használati esetek technikájával összegyűjthetjük és áttekinthetjük az alkalmazással szemben támasztott legfontosabb követelményeket. Első lépésként meghatározzuk az alkalmazáshoz kapcsolódó külső szereplőket, majd sorra megadjuk, hogy milyen funkciókon („interakciókon") keresztül történik ez a kapcsolódás. Ábrázolhatjuk a használati esetek lényegesnek ítélt alváltozatait, kiemelhetünk egyes résztevékenységeket, illetve jelölhetünk különböző megvalósítási módokat.

Jacobson

A *„használati eset"* `(use case)` elnevezés Jacobson-tól származik, aki társaival együtt először dolgozta ki részletesen ezt a technikát. A korai objektumorientált módszertanok nagy hiányossága volt, hogy nem adtak eszközt a követelmények áttekinthető ábrázolására, hanem erre a célra többnyire formalizálatlan szöveges dokumentumokat használtak. Az OO módszerek rövid idő alatt átvették a használati esetek egyszerű és látványos megoldását, ami azért is különös, mert ezt a technikát nem tartották objektumorientáltnak. A Rational Unified Process a fejlesztés teljes folyamatának ütemezését a használati esetek segítségével végzi el.

RUP

A *használati eset diagram* `(use case diagram)` az UML egy diagramtípusa. A jelöléseket a ***Q*** modellező módszer lépéseit követve tekintjük át.

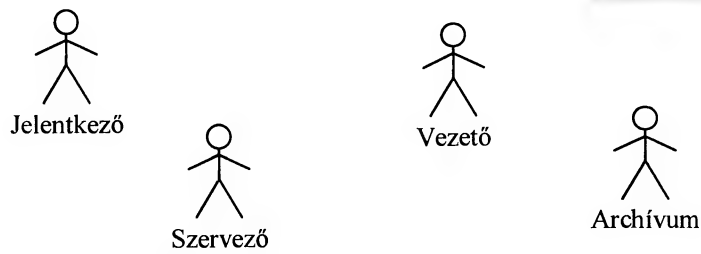
Aktor

Az alkalmazással szemben támasztott követelmények összegyűjtésének egy lehetséges kiindulópontja, ha meghatározzuk, hogy a rendszer határain kívül eső, „külső" elemek milyen módon kapcsolódnak a rendszerhez. A kapcsolódás lehetséges módjait *aktoroknak* `(actor)` nevezzük. Példaalkalmazásunkban ilyen aktor a `Jelentkező`, a `Vezető`, a `Szervező`, sőt, külső rendszerként az `Archívum` is.

jelölés

A használati eset diagramokon az aktorok jele a „pálcikaember", mely alá az aktor elnevezését írjuk. Az UML az aktorok elnevezésére egyes számban megadott főnevet vagy rövid főnévi szerkezetet (például `Oktatási vezető`) ajánl, mely első betűje nagybetű.

23. Ábra: Aktor



A fogalomszótárban — a modell más elnevezett elemeihez hasonlóan — célszerű egy-két bekezdésben összefoglalni a felvett aktorok alkalmazásban betöltött szerepét.

> ***Vezető.*** A tanfolyamokkal kapcsolatos irányító, menedzseri tevékenységeket látja el, így meghirdethet vagy különleges ok miatt (pl. oktató betegsége) lemondhat egy tanfolyamot.
>
> ***Szervező.*** A tanfolyamokkal kapcsolatos napi szervezői munkát látja el és figyelemmel kíséri a jelentkezéseket.

Az aktor egyszerűen egy «`actor`» sztereotípiájú osztály, amely ikonja a „pálcikaember", így az aktor elnevezése egy téglalapban is megjeleníthető. Új sztereotípiák felvételével, majd azokhoz ikonokat rendelve emberközelibbé tehetjük a használati eset diagramokat.

aktor ikonja

24. Ábra: Ikonok

Az UML-ben az aktor egy szerepet jelent, melyet több konkrét szereplő is eljátszhat, például vezetői jogkörrel az oktatási vezető és helyettese is rendelkezhet. Ugyanakkor egy konkrét szereplő több szerepben is megjelenhet, így egy kisebb cég esetén a `Vezető` és a `Szervező` ugyanaz a személy is lehet.

szerep

Az aktor jelenthet bizonyos felhasználókat, de lehet az alkalmazásunkkal kapcsolatban álló külső rendszer is.

Első lépésben nem szükséges az összes aktort azonosítani, elegendő csupán a legfontosabbakat.

Használati eset

A legfontosabb aktorok ábrázolásával már rendelkezésünkre áll egy használati eset diagram. A rendszer körvonalazását úgy folytathatjuk, ha minden egyes aktort sorra véve meghatározzuk, hogy azok mely funkciókon keresztül kapcsolódnak a rendszerhez. Ez úgy is megtehető, ha magunkat az egyes aktorok helyébe képzeljük. Üljünk be először a `Vezető` kényelmes bőrfotelébe, aki modern számítógépén éppen meghirdet egy néhány hét múlva esedékes tanfolyamot vagy az oktató megbetegedéséről tudomást szerezve lemondja a következő héten tartandót. Kevésbé kényelmes székében kevésbé modern számítógépe előtt ülve a `Szervező` figyelemmel kíséri a jelentkezéseket és a tanfolyamot a megtartása előtt néhány nappal

véglegesíti vagy elegendő jelentkező hiányában lemondja. A döntésről a jelentkezőket automatikus e-mail üzenet értesíti.

használati eset

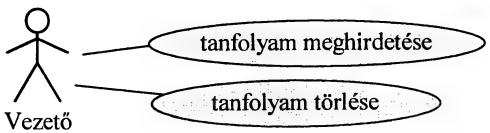
A rendszer külső kapcsolódási pontjait, más néven kívülről látható funkcióit *használati eseteknek* (`use case`) nevezzük. A használati eset például egy összetett párbeszéd (interakció) a felhasználó és az alkalmazás között.

jelölés

Az UML a használati eseteket ellipszis alakzattal jelöli, melyben az elnevezést középre igazítva adjuk meg. Névként az UML rövid aktív igei szerkezetet ajánl, mely első betűje nagybetű. A ***Q*** módszer kisbetűs elnevezéseket javasol.

Az aktorok és a funkciók között húzott vonallal jelöljük, hogy az egyes aktorok milyen használati esetekkel kapcsolódnak a rendszerhez. A kapcsolatot az alkalmazással lefolytatott kommunikáció lehetőségeként értelmezzük.

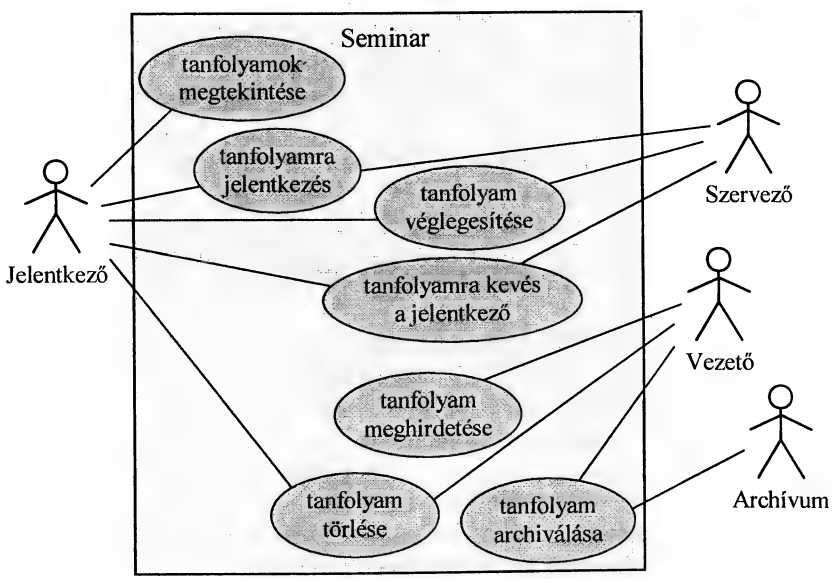
25. Ábra: Használati eset



Fogalomszótárunkat célszerű bővíteni a felvett használati esetek rövid összefoglalásával, értelmezésével.

A használati eseteket megadhatjuk egy, az alkalmazást jelölő téglalapban is, melyet az alkalmazás nevével jelölünk el.

26. Ábra: Alkalmazás használati esetei



Példánkban a `tanfolyamra kevés a jelentkező` használati esetnek nem megfelelő az elnevezése, ugyanis az nem a funkcióra, hanem annak feltételére utal. Célszerűbb helyette egy másik elnevezést választani, amely lehet például a `tanfolyam lemondása`.

Aktor és használati eset

Az UML megengedi, hogy a kapcsolat típusát a vonalak végére helyezett nyílhegyekkel jelöljük. Értelmezésében a nyíl az információtovábbítás irányát ábrázolja, míg az irányítatlan vonal az „oda-vissza" adattovábbítást, azaz információcserét, kommunikációt jelöli. Sajnos ez a megközelítés ellentmond a más diagramokon felvett irányított vonalak értelmezésének (a „navigációs iránynak").

vezérlés (használat) jelölése

Az UML gyakorlatában az aktorokat és a használati eseteket kizárólag folytonos vonallal vagy nyíllal kötik össze. A ***Q*** módszer ettől eltérve, de még a szabvány keretein belül maradva a szaggatott vonalú nyilak használatát javasolja, mely nem az információtovábbításra, hanem a vezérlésre (a „használatra") utal.

folyamat beindítója

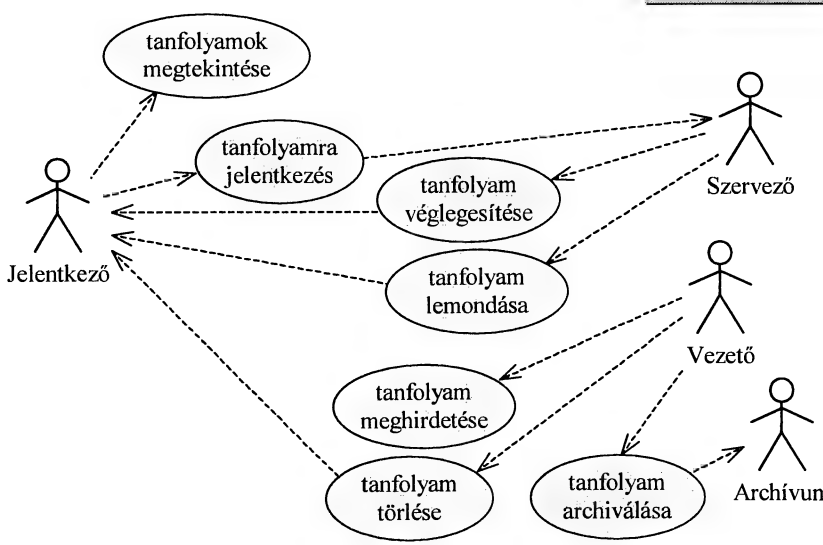
- Ha az aktor indítja be a folyamatot, akkor a nyílhegy a használati esetben végződik. Ugyanezt a jelet használjuk, ha a beindítást követően „oda-vissza" információtovábbítás zajlik le.

folyamat végrehajtója

- A használati esetből kiinduló szaggatott vonalú nyíl az aktorban végződik, ha a használati eset indít be az aktorban valamely folyamatot. A használati eset egyoldalúan kényszerítheti az aktort valamely tevékenységre, ekkor az aktor a folyamat *végrehajtója*, más néven tárgya, „szenvedő alanya". „Oda-vissza" nyilat használunk, ha nem az aktor indítja el a funkciót, de a folyamat aktív *közreműködője*, így beavatkozhat annak menetébe.

közreműködő

27. Ábra: Vezérlés



Az adatközlést (UML), illetve a vezérlést alapul vevő két megközelítés között csak akkor van eltérés, ha az adatközlés iránya eltér a vezérléstől. Ilyen például a lekérdezés, amikor az aktor kérésére (→) az alkalmazás szolgáltat adatokat (←).

Aktorok általánosítása és pontosítása

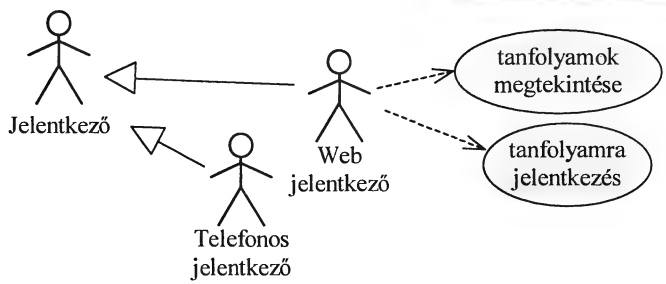
Néhány használati eset összegyűjtése után célszerű felülvizsgálni a modellünket, hogy az aktoroknak léteznek-e említésre érdemes alváltozatai. A különböző változatok az alkalmazásunkhoz eltérő módon kapcsolódhatnak.

Ha nem csak a Web-en, hanem a fax-on vagy telefonon keresztül történő jelentkezéseket is figyelemmel kívánjuk kísérni, akkor ezt a diagramunkon is jelölhetjük. A telefonon keresztül történő jelentkezés valószínűleg a szervező közvetítését igényli.

jelölés

Az UML az általánosítás-pontosítás viszonyt háromszögben végződő vonallal jelöli, ahol a háromszög az általánosabb elem felé mutat.

28. Ábra: Aktor pontosításai



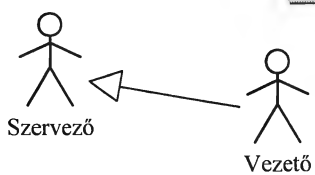
modell ellenőrzése

A lényeges alváltozatok kigyűjtése mellett ellenőrizzük a már korábban felvett aktorokat is, hogy azok közvetlenül vagy közvetve nincsenek-e általánosítás-pontosítás viszonyban.

Két aktor között általánosítás viszony állhat fenn, ha az egyik (az általánosabb) összes használati esetéhez a másik is ugyanolyan módon kapcsolódik. Ebben az esetben vizsgáljuk meg, hogy a valóságban is létezik-e a viszony, tehát az általánosabb szerepkörbe a speciálisabb aktor bármely konkrét felhasználója is behelyettesíthető.

Ha a példánkban úgy döntünk, hogy a `Szervező` tevékenységeit a `Vezető` számára is elérhetővé tesszük, akkor ez azt jelenti, hogy a `Vezető` is rendelkezni fog szervezői jogkörrel, azaz a `Vezető` is `Szervező`.

29. Ábra: Aktor általánosítása



A közvetlen általánosítás-pontosítás viszonyok jelölése mellett azt is célszerű megvizsgálni, hogy bizonyos aktorok valamilyen szempontból nem tartoznak-e egy közös típusba. Ha kettő vagy több aktor azonos használati esetekkel és azonos módon kapcsolódik a rendszerhez, akkor ez utalhat arra, hogy az aktoroknak létezik egy közös általános változata. Ha a valóságban ténylegesen fennáll ez a viszony, akkor keressük meg és nevezzük el az általános aktort, majd a diagramon az aktorokat az általános pontosításaiként, változataiként jelöljük.

Az általánosítás-pontosítás viszonyok megkeresésével egyszerűbbé és konzisztensebbé tehetjük modellünket, mivel az általános aktorokhoz kiemelhetjük a közös kapcsolódási módokat és a változatok esetén áttekinthetőbbé tehetjük az eltéréseket.

Változatok (extends)

A használati esetek összegyűjtésekor kezdetben célszerű a legfontosabb, leghangsúlyosabb, normálisnak (*„happy day"*) tekintett eseteket megkeresni. Ezután vizsgáljunk meg minden egyes esetet, hogy létezik-e annak külön figyelmet érdemlő, eltérő működése. Az eltérést speciális körülmények vagy hibák, például hibás adatok is okozhatják.

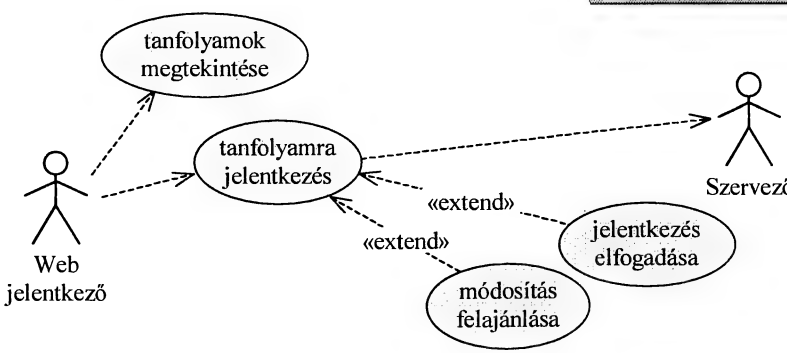
eltérő, speciális vagy hibás működés

UML jelölésekkel a diagramon az egyes változatokat önálló használati esetként adjuk meg, melyből az alapesetig «extend» sztereotípiával jelölt szaggatott vonalú nyilat húzunk.

jelölés

A `Seminar` példaalkalmazásunkban lehetséges, hogy egy tanfolyamra a szabad helyeknél több fő kíván jelentkezni. Nem tehetjük meg, hogy ekkor egyszerűen figyelmen kívül hagyjuk az érdeklődést. Kézenfekvő megoldásnak tűnik, hogy módosításként felajánljuk a következő tanfolyamot vagy a még szabad helyeket. A módosítás során dönthetünk arról, hogy a `Szervezőt` csak a sikeres jelentkezésekről informáljuk, vagy a sikertelenekről is

30. Ábra: Változatok



A változatok az alapeset működéséhez bizonyos többletet adnak. Azokat a használati eseteket jelöljük kiterjesztésként, amelyek az alapeseten belül opcionálisak, azaz nem feltétlenül, vagy csak bizonyos feltételek esetén hajtódnak végre.

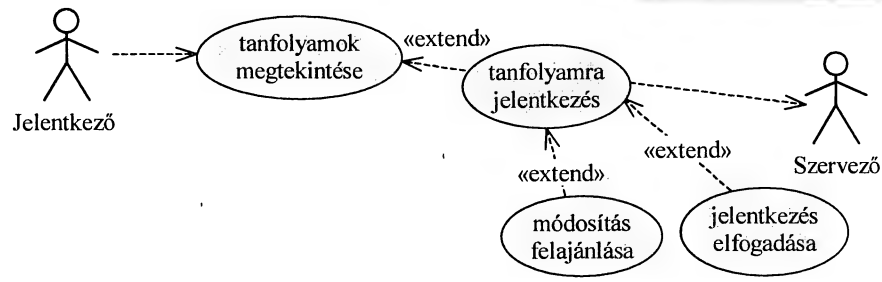
modell ellenőrzése

A használati esetek lényeges változatainak felvétele mellett a kiterjesztés kapcsolat alapján ellenőrizzük, és ha szükséges, módosítsuk modellünket:

- Ha egy használati eset egy másiknak a változata, akkor a normál működéshez az alesetet kapcsoljuk kiterjesztésként.
- Ha találunk két olyan használati esetet, amelyek egy még fel nem vett használati eset változatai, akkor határozzuk meg ezt a normál esetet, és ahhoz kapcsoljuk kiterjesztésként a két funkciót.

A példánkat jobban megvizsgálva észrevehetjük, hogy a tanfolyamra jelentkezés valójában az érdeklődés speciális változata.

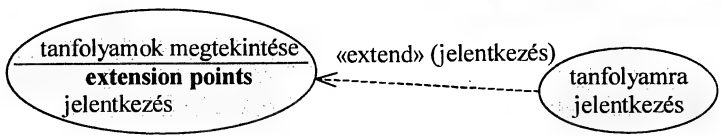
31. Ábra: Változatok



helyzet-jelentések

A kiterjesztések úgy képzelhetők el, hogy az alap használati eset a működésének bizonyos pontjain „jelenti" a fennálló helyzetet, melyet figyelve a változatok közbeszúrhatják a helyzetnek megfelelő tevékenységüket. Ha például a `tanfolyamok megtekintése` egyszer csak jelentkezésbe „fajul el", akkor a `tanfolyamra jelentkezés` használati eset lép közbe, hogy lebonyolítsa a szükséges adminisztrációt.

32. Ábra: Kiterjesztési pont



kiterjesztési pont

Az UML jelöléseivel a használati eset alakzatának alsó részében egy `extension points` felirat alatt soroljuk fel a kiterjesztési pontokat, „helyzet-jelentéseket". Az «extend» után zárójelben megadhatjuk, hogy a kiterjesztés mely kiterjesztési pontra vonatkozik.

Egy használati eset működésén belül ugyanaz a kiterjesztési pont többször is előfordulhat, illetve adott elnevezésű kiterjesztési pontra több változat is rákapcsolódhat.

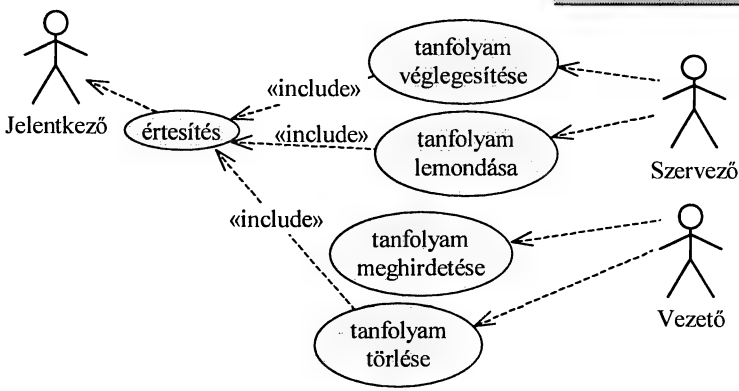
Részfunkció (include)

szükséges résztevékenység

Egy használati eset működéséhez szükséges lehet olyan részfunkció, amelyet hangsúlyozni szeretnénk, például azért, mert a megszokottnál több erőforrást (időt és pénzt) igényel a megvalósítása. Ugyancsak részfunkcióként ábrázolhatjuk a több használati eset működésében megismétlődő közös részletet.

Példánkban a `Jelentkező` értesítésének részfunkciója több használati esetben is megjelenik.

33. Ábra: Részfunkció



jelölés

UML diagramokon a részfunkciós kapcsolatot úgy ábrázoljuk, hogy az alapesetből a lényegesnek tartott részfunkcióig `«include»` sztereotípiával jelölt szaggatott vonalú nyilat húzunk.

„használat"

A ***Q*** modellező módszer a szaggatott vonalú nyilat egy általános „használat" jelöléseként értelmezi. A használati esetek között az `«include»` sztereotípiát alapértelmezettnek tekinti, melyet nem szükséges kiírni.

A változatokhoz hasonlóan a részfunkciók is a modellünk részletezésének eszközei. Sorra véve minden egyes használati esetet gondoljuk át, hogy annak létezik-e lényeges résztevékenysége, illetve ellenőrizzük, hogy egy már felvett használati eset nem valamely más használati eset részfunkciója.

Használati esetek általánosítása

megvalósítás változatai

Az aktorokhoz hasonlóan a használati eseteknél is megadható az általánosítás-pontosítás viszony. Egy használati eset pontosítása olyan funkciót jelöl, amely az általános helyére behelyettesíthető. Pontosításként jelölhetjük az általános megvalósítási lehetőségeit, illetve általánosként ábrázoljuk több használati eset hasonló jellegét. Például egy `jelentés továbbítása` történhet Interneten vagy két gép közötti közvetlen telefonos kapcsolaton keresztül.

jelölés

Az UML az általánosítás-pontosítás viszonyt a pontosított elemtől az általánosig húzott, háromszögben végződő vonallal jelöli.

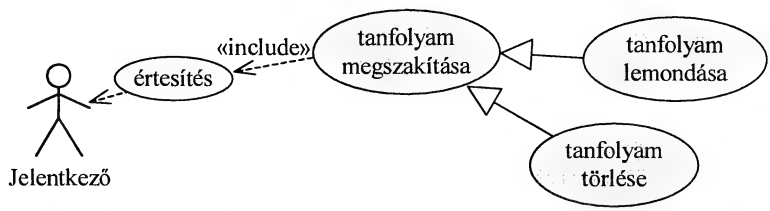
modell ellenőrzése

A használati eseteknek az általánosítás alapján történő ellenőrzését a többi viszony ellenőrzéséhez hasonlóan végezzük el.

- Minden egyes használati esetet sorra véve vizsgáljuk meg, hogy annak létezik-e említést érdemlő pontosítása.
- Ellenőrizzük, hogy
  - két használati eset nincs-e általánosítás-pontosítás viszonyban. Erre a viszonyra utalhat, ha az egyik (az általánosabb) használati eset összes kapcsolatával a másik is rendelkezik. Ekkor mindig vizsgáljuk meg, hogy a valóságban is fennáll-e a viszony.
  - adott használati eseteknek található-e általánosabb változata. Erre utalhat, ha a használati eseteknek léteznek közös kapcsolataik.

Példánkban a tanfolyam lemondása és törlése egyaránt azt jelenti, hogy a meghirdetett tanfolyam nem lesz megtartva. Ha a közös jellegzetességet lényegesnek ítéljük, akkor azt a diagramunkon általánosításként ábrázolhatjuk.

34. Ábra: Használati esetek általánosítása



Változat, részfunkció vagy általánosítás?

Az «extend», «include» kapcsolatok, illetve az általánosítás jelölései egyaránt a használati eset diagram részletezésének az eszközei.

ismétlődő működés kiemelése

- Az «include» kapcsolat olyan *részfunkciót* jelöl, amely az alap használati eset működéséhez nélkülözhetetlen, így annak szerves része. Az UML szerzőinek hasonlatát követve a részfunkció működését *kiemeljük* az alapesetből.

működés beszúrása

- A normál működés bizonyos pontjain adott feltételek mellett bekövetkező, de attól részben független *kiegészítéseket* az «extend» kapcsolattal jelölhetjük. Az UML szerzőinek hasonlata alapján a kiterjesztés működését *beszúrjuk* a kiindulásként tekintett működésbe.

megvalósítási lehetőség

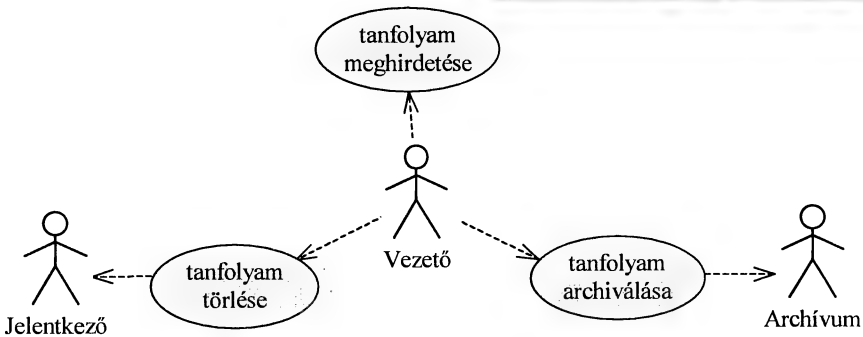
- Egy *általános* használati eset adott pontosítása egy *megvalósulási lehetőséget* ír le, így bármely pontosított változat felhasználható ott, ahol szükségesként csak az általános változatot jelöljük meg. Az általánosítás viszony így a jelleget mutatja meg, míg az «extend» és «include» módon összekapcsolt elemek a működésen belüli működést határoznak meg.

Az «include» megfogalmazható *szükséges* résztevékenységként, az «extend» pedig *lehetséges* kiegészítésként. Ezért, ha a használati eseten belül egy másik használati eset biztosan („feltétel nélkül") végrehajtódik, akkor azt «include» módon kell az alapesethez kapcsolni, míg az «extend» egy lehetséges útirányt, a normálnak tekintett viselkedés elfajulását, változatát jelöli.

Diagramok kialakítása

Az UML jelöléseivel történő rendszerszervezés napi gyakorlatában a modell elemeit általában csillag alakban szokták elrendezni. Gyakori, hogy középre helyezik az aktort, a használati eseteket pedig egy képzeletbeli körívre, egymástól egyenlő távolságra.

35. Ábra: A „csillag" elrendezés



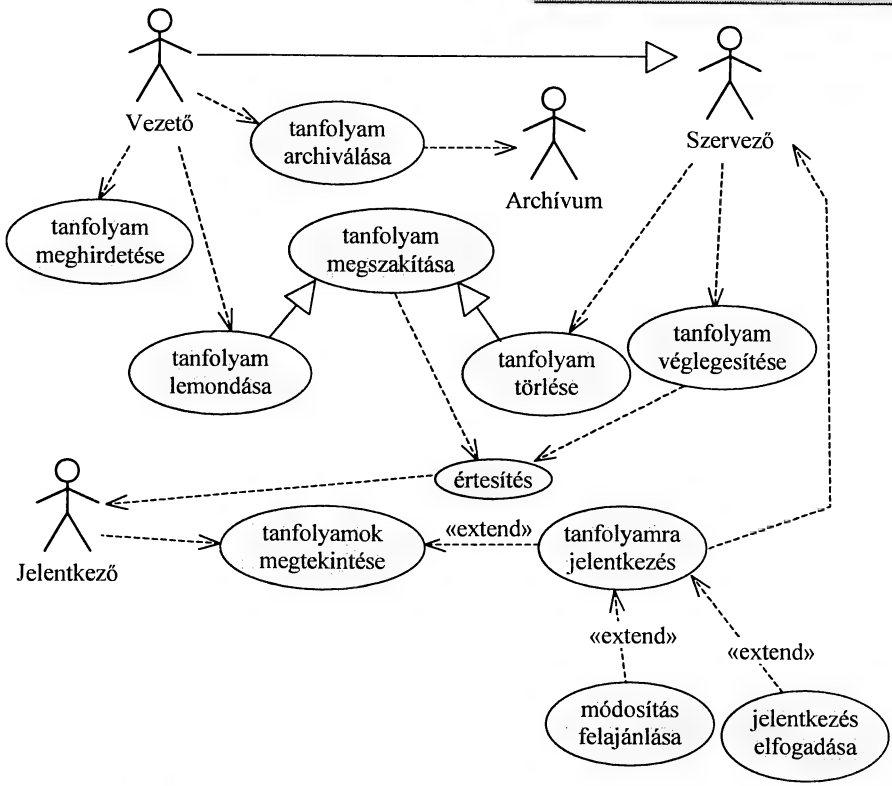
A ***Q*** modellező módszer a diagramok kialakítására általános szabályként az olvasási sorrendet (fentről-le és azon belül balról-jobbra) ajánlja. A hangsúlyosabb elemeket (például az alapelemeket) így fent vagy balra, a kisebb fontosságúakat (általában az alelemeket) pedig lent vagy jobbra adjuk meg. Jegyezzük meg, hogy ez az általános kialakításra vonatkozó javaslat, melytől el lehet és el is kell térni, ha a konkrét helyzet ezt kívánja.

olvasási sorrend

Használati eset diagramokra a ***Q*** módszer ajánlását a következőképpen alkalmazhatjuk:

- Az aleseteket (az «extend»-el vagy «include»-al kapcsolódó eseteket, illetve a pontosításokat) az alapesettől számítva lefelé vagy jobbra célszerű elhelyezni.
- Aktorok elhelyezése:
  - Az intenzív használókat, például az érdeklődőket a diagram baloldalán helyezzük el.
  - A kontroll-jellegű (adminisztrátori szerepű) aktorok felül, a fontossági sorrendnek megfelelően balról jobbra.
  - A szolgáltató, illetve kiszolgáló elemek a diagram felső-jobb oldalán, a jobboldalán, vagy a jobb-alsó, esetleg az alsó részén helyezkedjenek el a fontossági sorrendjüknek megfelelően.

36. Ábra: Diagram kialakítása



Felhasználó céljai és az interakciók

A használati esetek egy egyszerűbb vagy összetettebb párbeszédet, azaz interakciót jelentenek. Fontos azonban, hogy különbséget tegyünk a rendszerrel történő interakciók és a felhasználó céljai között. A `Szervező` a tanfolyamot sem véglegesíteni, sem lemondani nem akarja, célját, a szervezéssel kapcsolatos mechanikus tevékenységek automatizálását (típusüzenetek összeállítása, stb.) azonban ezeken a funkciókon keresztül tudja megvalósítani.

felhasználói cél az elsődleges

Elsődlegesnek természetesen a felhasználó céljait kell tekintenünk. A használati eseteket úgy kell összeválogatnunk, hogy azok lehetővé tegyék ezeket a célokat.

Az alkalmazás áttekintésében, mely első változatát a fejlesztés kezdetén fogalmaztuk meg, az alkalmazás környezete, azaz a szituáció ismertetése mellett a felhasználói célokra helyezzük a hangsúlyt.

jelölés (UML)

A felhasználói célokat a diagramon egy `«requirements»` sztereotípiájú megjegyzésen adhatjuk meg. Ha egy modellelemnek a megvalósítás szempontjából vett célját szeretnénk megadni, akkor `«responsibilities»` sztereotípiájú megjegyzést használjunk. Megjegyzések esetén az UML csak ezt a két sztereotípiát definiálja.

Használati eset diagramok

A használati esetek alapvető fontosságúak a követelmények elemzésben. Segítségükkel rajzolhatjuk meg a rendszer határait, hogy mely funkciók kerüljenek be a kifejlesztendő alkalmazásba és melyek ne. Ez utóbbi szemléltetéseként készíthetünk olyan diagramokat, melyeken az egyértelműség kedvéért egy vastag vonallal áthúzzuk azokat a funkciókat, melyekre nem terjed ki az alkalmazásunk.

rendszer határai

A használati eseteket úgy kell összeválogatnunk, hogy segítségükkel teljesíteni tudjuk az áttekintésben megfogalmazott összes felhasználói célt.

A kezdők számára az egyik leginkább zavarba ejtő kérdés, hogy milyen részletes legyen a használati eset diagramunk. Eredeti könyvében Jacobson egy 10 emberéves munka esetén mindössze 20 használati eset rögzítését ajánlja (az *emberév* a résztvevők számának és azok munkával töltött idejének a szorzata). Fowler és Scott hasonló helyzetben 100-nál több használati eset megadását tartja célszerűnek, azaz egy használati eset kb. egy ember egy hónapos munkáját jelenti. Részletesebb használati esetekkel a fejlesztés jobban behatárolható és kisebb kockázattal ütemezhető. Túl sok használati esetet felvéve azonban könnyen áttekinthetetlenné válhat a diagramunk.

részletezés

Az értelmezhetőséget úgy biztosíthatjuk, ha egy áttekintő, a legfontosabb használati eseteket tartalmazó diagramot készítünk, az egyes alesetekhez kapcsolódó változatokat, illetve résztevékenységeket pedig külön diagramokon ábrázoljuk. Szemléltetésül olyan diagramokat is készíthetünk, amelyre csak egy aktort és az ahhoz kapcsolódó néhány használati esetet vesszük fel.

aldiagram

Minden diagram alatt célszerű megadni a fogalomszótárnak a diagram elemeire vonatkozó részletét.

A követelmények összegyűjtése során csak olyan használati esetet szabad felvenni, amely a felhasználó által is látható, így ne ábrázoljunk a megvalósítással kapcsolatos technikai részleteket. A diagramokkal az a célunk, hogy a kifejlesztendő alkalmazásról pontosabb képet állítsunk össze.

Fowler és Scott említett művének megfogalmazása szerint (9. oldal) a rendszerfejlesztés legnagyobb kihívása a „*megfelelő* rendszer felépítése — amely találkozik a felhasználó igényeivel és megindokolható költségű". Sajnos nagyon sok fejlesztői csoport számára ez csak egy elérhetetlen idea, mivel „ad hoc módszer" alkalmazása esetén a felhasználói igények kontrolálatlanul egyre magasabbra tornyosulhatnak, miközben a legvégső határidő mind későbbi időpontokra tolódik.

A követelmények körvonalazása kettős célt szolgál. A felhasználók pontosabb képet alkothatnak a kifejlesztendő rendszerről, így tudni fogják, hogy mely funkciókkal és lehetőségekkel rendelkezik majd az alkalmazás és melyekkel nem. A másik oldalon álló fejlesztők a követelmények tisztázásával megismerik a megrendelők igényeit és meg tudják becsülni — sőt, meg is tudják indokolni — a fejlesztés erőforrásigényét, az idejét és a bekerülés költségét.

követelmények körvonalazása

A használati eset diagramjaink részletezettsége akkor megfelelő, ha

- a megrendelő számára egyértelmű, hogy mely funkciók, változatok, részfunkciók és megvalósítási módok kerülnek be az alkalmazásba és melyek nem,
- másrészt, a fejlesztők meg tudják becsülni a fejlesztés erőforrásigényét.

erőforrás-igény becslése

Az erőforrásigény becsléséhez felhasználhatjuk a Rational Unified Process által javasolt módszert, mely szerint a használati esetekhez időt (hány ember hány hetes munkáját igényli) és pénzbeli bekerülési költségeket rendelünk. A kettő nem feltétlenül van arányban egymással, mivel bizonyos használati esetek megvalósítása szakértő közreműködését igényelheti.

A használati esetekhez költségeket rendelve megkönnyíthetjük a felhasználó döntését, hogy a hasznosság és a költségek összehasonlításával kiválaszthassa a valóban szükséges funkciókat és megvalósítási módokat.

fejlesztés ütemezése

Az összegyűjtött használati esetek a fejlesztés menetének ütemezésére is felhasználhatók. Iteratív módszer esetén meghatározhatjuk az egyes iterációs ciklusok végén elérendő eredményeket, az inkrementumokat.

Példánkban az első iteráció céljának célszerű a meghirdetett tanfolyamok és a jelentkezések nyomon követését választani, mivel ez központi szerepű az alkalmazásban. A második célja lehet a Web-en keresztül történő megtekintés és jelentkezés megvalósítása, míg az archiválási funkciót az utolsó iterációra halaszthatjuk.

A használati eset diagramok tekinthetők a rendszerrel szemben támasztott követelmények térképeként, a kifejlesztendő alkalmazás funkcióinak grafikus tartalomjegyzékeként.

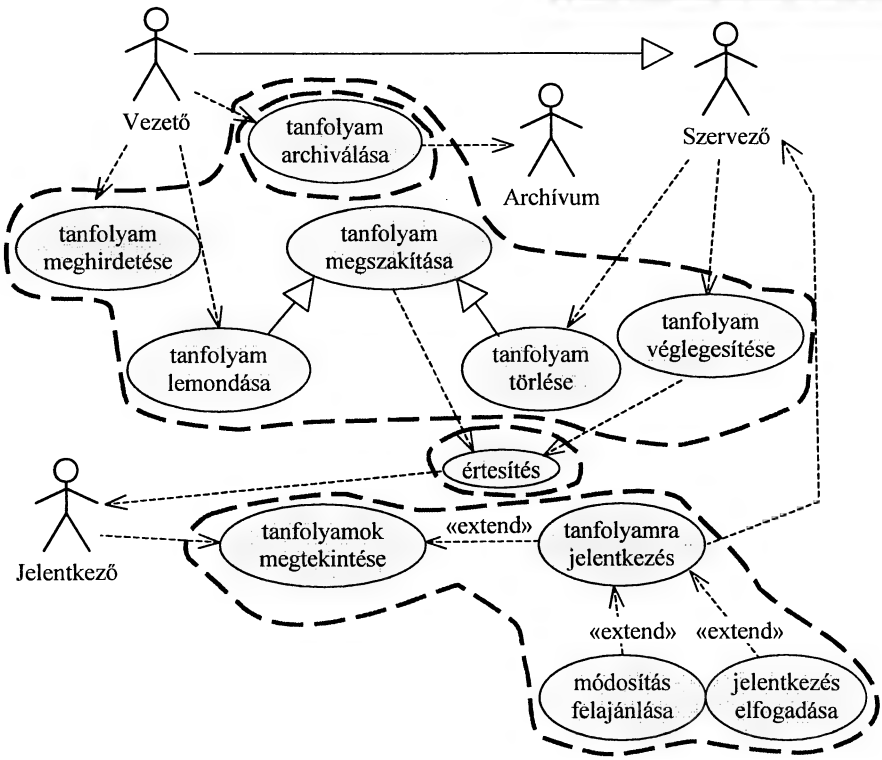
Csomagok

Csomagok segítségével a modellünk szorosabban összetartozó építőelemeit magasabb szintű egységekbe csoportosíthatjuk. Az összetartozó egységek lefedhetnek egy funkciókört, megvalósíthatnak egy felhasználói célt vagy biztosíthatják a szükséges technológiai hátteret. A csomagok jelölésével az alkalmazásunk részei áttekinthető hierarchiába szervezhetők. A csomagokra bontás a fejlesztés ütemezésének és a tesztelésnek is a kiindulópontja.

A használati esetek összegyűjtése során észrevehetjük, hogy egyes használati esetek az ellátott funkciók alapján csoportokba sorolhatók. Gyakori, hogy adott felhasználók adott funkciócsoportokon keresztül kapcsolódnak a rendszerhez, így az aktorok is sugallhatják a használati esetek csoportosítását. *csoportosítás*

A ***Q*** modellező módszer ajánlását követve a legfontosabb használati esetek felvétele után egyszerűen rajzoljuk körbe az összetartozó funkciókat, majd ábrázoljuk csomagokként az egyes csoportokat.

37. Ábra: Funkciócsoportok



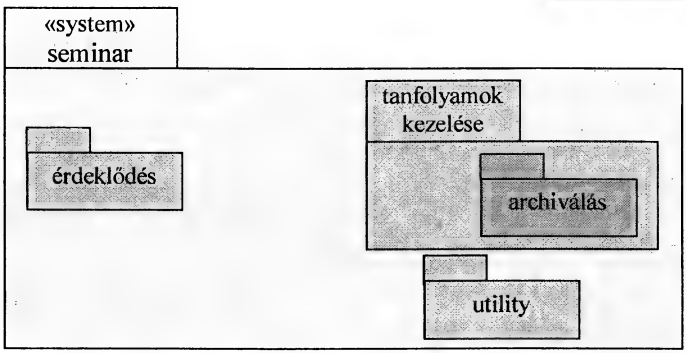
Csomag (package)

A funkciócsoportok meghatározásával a rendszerünk részekre, modulokra vagy más néven csomagokra bontható. A *csomag* `(package)` a rendszer építőelemeinek magasabb szintű egységekbe csoportosításának az eszköze.

jelölés

A csomag jele a „címkézett téglalap". A címkén vagy az alakzaton belül adjuk meg a csomag nevét, mely sztereotípiával (a név felett) és megszorítással (az elnevezés alatt) is kiegészíthető.

38. Ábra: Csomagok

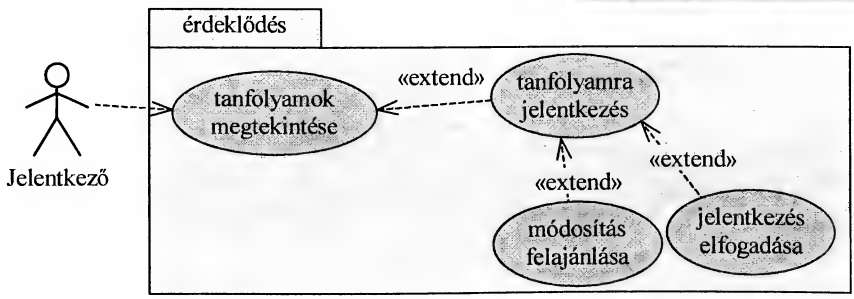


lényegi tartalom

Részletesebb ábrázolás esetén a nevet a címkére írjuk és a téglalapon belül megjelenítjük a csomag *lényegi tartalmát*, mely megadható

- *szövegesen*, felsorolva a fontosabb alkotóelemeket (használati esetek csoportosítása esetén ez nem szokásos),
- *grafikusan*, így a csomag tartalma lehet egy másik diagram, melyen például a fontosabb használati eseteket vagy belső csomagokat ábrázoljuk.

39. Ábra: Belső diagram



«system»

A legfelső csomag kivételével a modell minden eleme pontosan egy csomaghoz tartozik, így a tartalmazási hierarchia egy fát alkothat. Az UML úgy tekinti, hogy létezik egy legfelső, `«system»` sztereotípiájú csomag, amelyet általában nem nevezünk el.

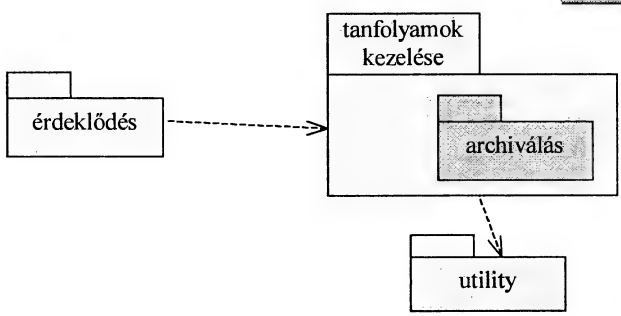
Az UML a csomagok esetén hat sztereotípiát definiál.

- «system» *(rendszer)*: a hierarchia tetején álló legfelső csomagot jelöli, azaz a teljes alkalmazást.
- «subsystem» *(részrendszer)*: az alkalmazáson belüli, részben független csomag.
- «model» *(modell)*: az alkalmazás egészének egy adott nézőpontból vett vázlatos képét mutatja.
- «framework» *(keretrendszer)*: az alkalmazás működtetéséhez szükséges általános elemeket tartalmaz.
- «stub» *(csonk)*: más csomag funkcióit utánzó helyettesítő csomag. Csonkot használhatunk például egy éppen fejlesztés alatt álló másik összetett csomag ideiglenes helyettesítésére vagy bizonyos hálózaton elérhető funkciók lokális (helyi) interfészeként.
- «facade» *(szemléltető nézet)*: más csomagok kiválasztott részeinek adott szempontból vett nézete, melyet általában szemléltetésként készítünk el.

Függőségek

Egy csomag valamely eleme használhatja egy másik csomag elemét. Az UML szóhasználatában ekkor a csomag függ a felhasznált másik csomagtól.

40. Ábra: Függőség



függőségi viszony

fordítási függőség

Az UML definíciója szerint két modellelem között *függőségi viszony* (dependency) van, ha az egyik definíciójának megváltozása a másik (a függő) elem megváltozását is eredményezheti. Ez a megfogalmazás a gyakorlatban fordítási függőséget jelent, azaz egy elem megváltozása esetén az attól függő elemeket újra kell fordítani. (A grafikus ábrázolás a programfejlesztők által jól ismert make parancs állományaiban megadható sorrendet jeleníti meg.)

Az UML a függőséget a függő modellelemből kiinduló szaggatott nyíllal jelöli. Gyakori, hogy a függőség típusát sztereotípia megadásával tovább pontosítjuk.

Két csomag esetén a függőség akkor áll fenn, ha a függő csomag valamely eleme függ a másik csomag egy elemétől.

{global}

A függőségek ábrázolását egyszerűsíthetjük a `{global}` megszorítás használatával. Úgy tekintjük, hogy a globális csomagoktól a rendszer minden más csomagja függ. A szakterület érték-jellegű típusait (például a `Cím` vagy `Dátum` típusokat) tartalmazó csomagot általában célszerű globálisként jelölni.

Névterület (namespace)

Minden modellelemet (a legfelső `«system»` csomag kivételével) pontosan egy csomagon belül adhatunk meg. A csomag egyben egy *névterület* *(namespace)* is, ezért a csomag határán belül az elemek közvetlenül hivatkozhatnak egymásra. Ugyanakkor a csomagon belül a modellelem elnevezésének egyedinek kell lenni, hogy az elemet egyértelműen azonosíthassuk.

elérési útvonal

Más csomagban lévő modellelemre az *elérési útvonal* megadásával hivatkozhatunk. Az UML az elérési útvonalat úgy képzi, hogy az egymást tartalmazó csomagok, majd végül a hivatkozandó elem nevét a „négyponttal" kapcsolja össze (például: `utility::értesítés`). (A *Q* módszer a „." használatát is engedélyezi.) A kiindulási pontról az első csomagnak elérhetőnek kell lenni.

«access» «import»

Az `«access»` sztereotípiájú függőséggel jelezhetjük, hogy a modellelem (például csomag) jogosan használhatja a hivatkozott csomag kívülről is látható elemeit. Ennél erősebb az `«import»` függőség, amely a névterületre mintegy „beemeli" a hivatkozott csomag elemeinek elnevezéseit úgy, hogy az elemekre a függő csomag az elérési út megadása nélkül is hivatkozhat.

A *Q* módszer a szaggatott nyilat egy általános „használat" jelöléseként értelmezi, amelybe az `«import»` (és egyben az `«access»`) jellegű hivatkozást is beleérti.

Csomag pontosítása

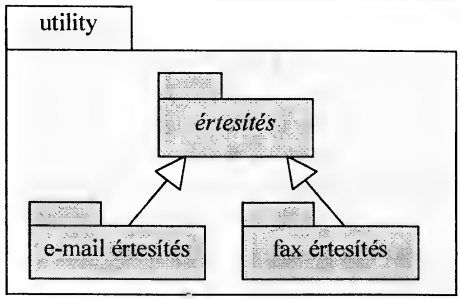
Egy csomag megfogalmazhat egy általános keretet, amelyet más csomagok tovább pontosíthatnak, illetve megvalósíthatnak.

Csomagok esetén is jelölhető az *általánosítás/pontosítás* viszony. Az általánosítás közvetve egy függőséget is jelent, a speciális függését az általánostól. A speciális felhasználja az általános definícióit, mivel ismernie kell a megvalósítandó általános keretet. Az `{abstract}` megszorítással vagy dőltbetűs elnevezéssel jelölhetjük azt a csomagot, amely önmagában nem alkalmazható, hanem mindössze egy általános használati módot ír le.

absztrakt csomag

Ha a példaalkalmazásunkat ki szeretnénk terjeszteni úgy, hogy a jelentkezőket a tanfolyam véglegesítéséről vagy megszakításáról ne csak e-mail-en, hanem fax-on keresztül is értesíthessük, akkor kézenfekvőnek látszik, hogy a `utility` csomagon belül egy általános `értesítés` csomagot `e-mail értesítés` és `fax értesítés` módon is megvalósítsunk.

41. Ábra: Csomag általánosítása/pontosítása



Csomagok használata

A csomagok alapvető fontosságúak minden nem-triviális alkalmazás esetén, mivel segítségükkel a modellünk elemeit áttekinthetőbb és kezelhetőbb egységekbe szervezhetjük.

A fejlesztés korai szakaszában, azaz néhány használati eset felvételét követően már célszerű csomagokra felbontani az alkalmazást. A modell finomodásával ezt a felbontást időnként ellenőrizzük és szükség esetén módosítsuk.

csomagokra bontás

A *Q* módszer ajánlása szerint minden csomagot tekintsünk részben független alkalmazásként, így készítsünk róla áttekintő leírást, melyben összefoglaljuk a csomag célját és a szituációt, az alkalmazáson belül elfoglalt helyét. Ezután készítsünk arról egy áttekintő és szemléltetésül néhány további használati eset diagramot.

önálló rész

A csomagokra bontás segítségével a fejlesztés menete is megtervezhető. Meghatározhatjuk a csomagok kidolgozásának időbeli sorrendjét, illetve egyes részeket különböző csoportok párhuzamosan is fejleszthetnek.

A csomagok egyben a későbbi tesztelés alapegységei is.

Csomagokat tartalmazó diagramok esetén az olvasási sorrend szabályát követve a függőségeket általában fentről lefelé, illetve jobbról balra vezetjük, a használót tekintve az összetettebb elemnek. A sorrendből így egyszerűen „kiolvasható" a vezérlés terjedése. Kivételt képeznek a pontosítások, amelyeknél a „függő" elemeket, azaz a lehetséges megvalósításokat ellentétesen helyezzük el. A technika fordítva is alkalmazható: ha természetesebbnek tűnik egy függőséget ellentétes irányba vezetni, akkor lehetséges, hogy a két csomag általánosítás/pontosítás viszonyban van egymással.

diagramok kialakítása

Általános szabály, hogy minimálisra kell csökkenteni a csomagok közötti függőségeket.

Rendszerek együttműködése

A Q modellező módszer az UML megszokott használatától eltérve, de még a szabvány keretein belül maradva ajánl néhány kiegészítést, melyek segítségével vázolható a rendszerek együttműködése. Egy új, «request» sztereotípiájú függőség segítségével modellezhetjük a közvetett és a megszakítás jellegű vezérléseket.

aktor és csomag

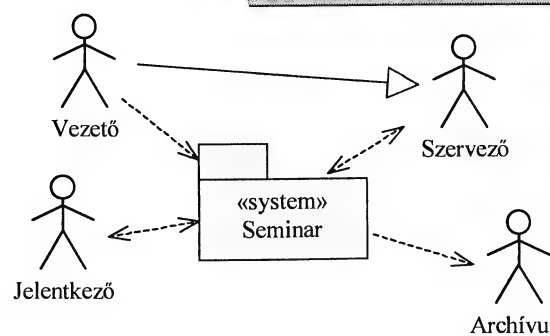
A ***Q*** módszer értelmezésében az aktorok illetve a csomagok hasonló módon kezelhetők. A módszer a hasonlóság miatt javasolja, hogy az aktort és használati eseteit szaggatott vonalú nyíllal kössük össze. Az aktor olyan külső rendszert jelent, amely az alkalmazás határain kívül helyezkedik el és ahhoz az alkalmazás csak közvetett, passzív módon, például adatok cseréjével kapcsolódik. Ezzel szemben az alkalmazás ismerheti egy csomag funkcióit, így például felhasználhatja azokat egy használati eset megvalósításának résztevékenységeként.

Ha elfogadjuk az aktorok és a csomagok közös jellegét, akkor a használati eset diagramokon a rendszerek együttműködését is felvázolhatjuk. A jelölésekkel még az UML szabvány keretein belül maradunk, bár eltérünk annak megszokott gyakorlatától.

alkalmazás külső kapcsolatai

A teljes alkalmazás is egy csomag, ezért egy használati eset diagramon ábrázolhatjuk az alkalmazás külső kapcsolatait. Sajnos, az UML nem engedélyezi, hogy egy szaggatott vonalú nyíl mindkét végén nyílhegy legyen, ezért ezt csak két, ellentétes irányú nyíllal rajzolhatjuk meg. A jelölés arra utal, hogy az alkalmazás és az aktor egyaránt tudomással van egymásról. Csomagok esetén lehetséges, hogy a belső tartalom ismeretében a függőségek egyes részekre pontosíthatók, így a nyilakat egyirányúvá bonthatjuk. Ebben az esetben a kétirányú nyíl több egyirányúnak a rövidítése. Ellenkező esetben a két csomagot együtt, egyetlen menetben kell lefordítani, mivel mindkettő „ismeri" egymást.

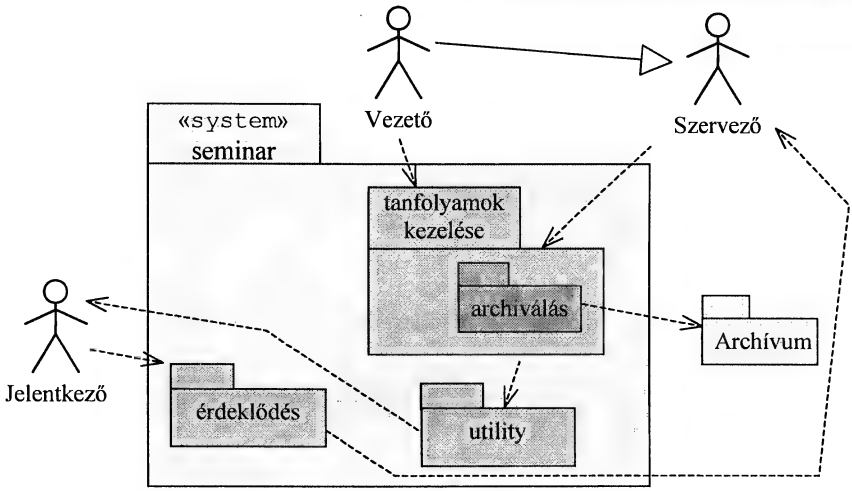
42. Ábra: Alkalmazás külső kapcsolatai



Példánkban az archívumot olyan külső rendszernek tekintettük, melyhez csak passzívan kapcsolódunk, például egy szöveges állomány adott helyre történő másolásával. Ha a kapcsolat aktív, így az archívum bizonyos funkcióira közvetlenül is hivatkozhatunk, akkor azt jelöljük az alkalmazáson kívül elhelyezkedő önálló csomagként.

Az alkalmazás «system» csomagján belül lényegi tartalomként elhelyezhetünk egy aldiagramot, így részletesebben is ábrázolhatjuk a külső kapcsolatokat.

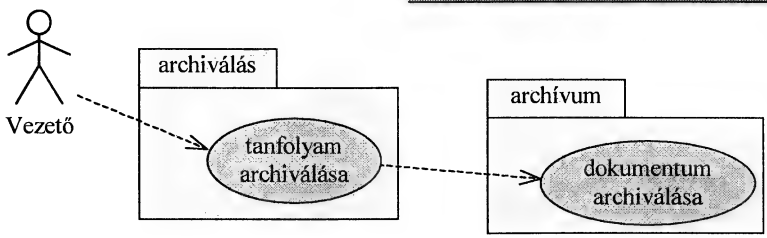
43. Ábra: Külső kapcsolatok pontosítása



A csomagokon belül újabb diagramot vehetünk fel, így megjeleníthetjük a csomag használati eseteit, illetve a használati esetekre konkretizálhatjuk a kapcsolódási módokat. A ***Q*** módszer a használati esetek közötti szaggatott vonalú nyilat alapértelmezésben «include» jellegű viszonynak tekinti, így azt nem szükséges külön leírni.

interakciók ábrázolása

44. Ábra: Csomag használati esete



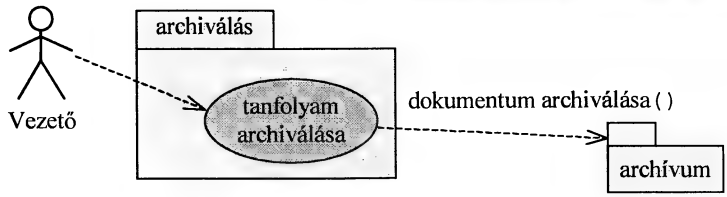
Korábban, az UML ajánlását követve egy téglalapban ábrázoltuk az alkalmazás használati eseteit. Az egységesebb jelölés érdekében itt is célszerűbb az alkalmazást nem téglalapként, hanem csomagként megjeleníteni.

A használati esetből egy csomagig vezető függőséggel jelölhetjük, hogy a használati eset valamilyen módon felhasználja, például kiterjeszti a csomag által szolgáltatott egy vagy több funkciót.

hivatkozott elem

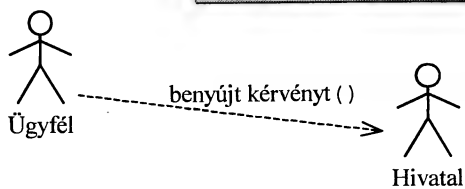
A ***Q*** módszer ajánlása szerint a csomagig vezető függőségen vesszővel elválasztva felsorolhatjuk a ténylegesen hivatkozott elemek elnevezéseit. Ha használati esetre hivatkozunk (nem pedig csomagra), akkor annak elnevezése után írjunk egy üres zárójelpárt (mintha az ellipszis alakzatra utalnánk). Kiterjesztés esetén a használati eset neve előtt adjuk meg az «extend» sztereotípiát, amelyet zárójelek között a kiterjesztési pont elnevezése követhet.

45. Ábra: Együttműködés csomaggal



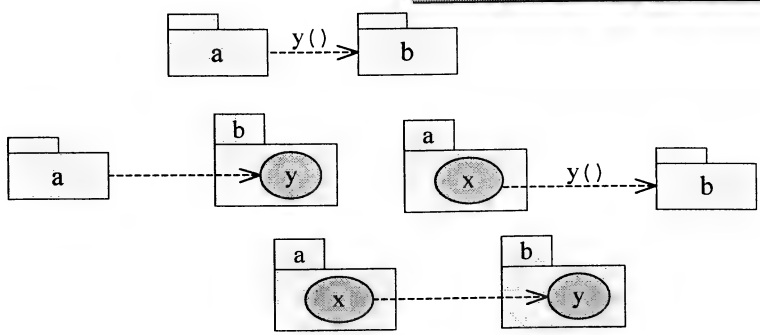
A technikával az aktorok közötti külső kommunikációt is ábrázolhatjuk.

46. Ábra: Aktorok közötti együttműködés



A jelölésekkel a csomagok elemeire történő hivatkozást, az együttműködést különböző részletezettségi szinteken, mégis egységes módon ábrázolhatjuk.

47. Ábra: Együttműködés ábrázolása



Közvetett vezérlés

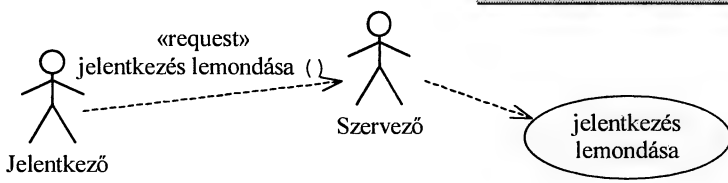
A `Seminar` példaalkalmazás használati eseteit megvizsgálva észrevehetjük, hogy a `Jelentkezőnek` nincs lehetősége lemondani egy előzetes jelentkezést. A problémára egy egyszerű megoldás, ha felveszünk a `Jelentkező` által elérhető `jelentkezés lemondása` használati esetet. A tanfolyamokat szervező cég szempontjából azonban nem feltétlenül szerencsés, ha a jelentkező közvetlenül („túl könnyen") lemondhatja a jelentkezését. Ehelyett célszerűbb olyan megoldást javasolni, melyben a jelentkező telefonon értesíti a szervezőt, aki az alkalmazásba már felvezetheti a változást.

A közvetett vezérlésre az UML nem rendelkezik külön jelöléssel. Hasonló szituációk modellezésére a ***Q*** módszer a `«request»` sztereotípiával megjelölt függőséget ajánlja, ahol a küldött kérést címkeként adhatjuk meg.

«request»

A közvetett vezérlést jelölhetjük két aktor közötti `«request»` kapcsolattal, ahol a küldött kérés a fogadó aktor által elért használati eset nevével egyezik meg.

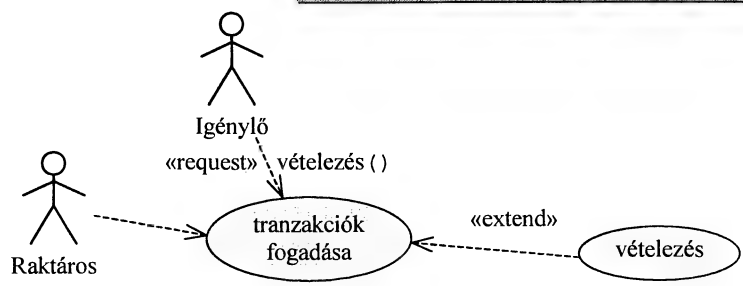
48. Ábra: Közvetett vezérlés



A `«request»` kapcsolattal használati eset közvetett vezérlése is jelölhető. Ekkor a küldött kérésnek vagy a használati eset által elért részfunkció, változat vagy egy kiterjesztési pont nevével kell megegyezni. Így modellezhetjük, hogy egy adott aktor által elindított használati eset alkalmas egy másik aktor kéréseinek fogadására.

használati eset közvetett vezérlése

49. Ábra: Használati eset közvetett vezérlése



Általánosan: a `«request»` kapcsolattal azt jelezzük, hogy egy aktor vagy egy használati eset normál „működésének" menetébe egy másik aktor vagy használati eset időlegesen közbeavatkozhat. A beavatkozás úgy történik, hogy a kérés hatására a fogadó (ha fogadásra képes állapotban van) végrehajt valamely tevékenységet,

majd visszatérhet a normál működéséhez, de félbe is szakíthatja azt. Például a `Szervező` a „normál működése" során egy `Jelentkezőtől` kapott telefon vagy e-mail hatására jelezheti az alkalmazásnak, hogy lemondtak egy jelentkezést, majd folytathatja napi tevékenységét.

A követelményelemzés szöveges dokumentumai

*A **Q** modellező módszer a követelmények pontosítására további szöveges dokumentumokat ajánl, melyekkel konkrét formába önthetjük a felhasználók igényeit. A kiegészítésekkel részletesebb képet alkothatunk a kifejlesztendő alkalmazásról, egyben azok a megrendelőkkel történő egyeztetésekre is felhasználhatók.*

Az alkalmazással szemben támasztott követelményeket tovább pontosíthatjuk néhány szöveges dokumentummal. A használati esetek összegyűjtését és csomagokba csoportosítását követően a `centre-out` elvnek megfelelően kiválasztjuk az első iterációban megvalósítandó legfontosabb használati eseteket és részletesebben megfogalmazzuk azok működését. A **Q** modellező módszer a követelmények pontosítására több szöveges dokumentumot is ajánl.

Alkalmazási példák

A használati esetek olyan funkcióelemek, amelyek elérhetővé teszik a felhasználói célokat. A követelményeket célszerű kiegészíteni olyan felhasználási, alkalmazási példákkal, amelyekkel szemléltethetjük, hogy a használati esetek milyen módon alkalmazhatók az egyes felhasználói célok megvalósítására.

felhasználói célok megvalósítása

50. Ábra: Alkalmazási példa

Tanfolyam Interneten történő meghirdetése, jelentkezés, archiválás:

- A `Vezető` *meghirdeti* a tanfolyamot az Interneten
- Meghirdetett tanfolyam esetén:
  - ➢ A `Vezető` valamely különleges ok miatt *lemondhat* egy meghirdetett tanfolyamot, ekkor a jelentkezőket értesíteni kell
  - A tanfolyamot a `Jelentkező` *megtekintheti*
    - ➢ A `Jelentkező` előzetesen *jelentkezhet* a tanfolyamra
      - A létszámtól függően az alkalmazás
        - *elfogadja* a jelentkezést, vagy
        - *módosítást* ajánl fel
  - ➢ Egy `Jelentkező` előzetes jelentkezése a `Szervező` közreműködésével *lemondható*
  - A létszámtól függően a `Szervező` *véglegesíti* vagy *lemondja* a tanfolyamot, melyről értesíti a `Jelentkezőket`
- A véglegesített tanfolyamot megtartják
- A tanfolyam időpontja után a `Vezető` *archiválhatja* a tanfolyamot

időbeli sorrend

Az alkalmazási példákkal a használati esetek logikai kapcsolatait, időbeli egymásutániságukat ábrázolhatjuk, így rögzíthetjük az érvényes sorrendeket.

tevékenység

Az alkalmazási példákat folyamatos szöveg helyett célszerű könnyebben olvasható pontokba foglalni. A pontokban általában tevékenységeket sorolunk fel, amely lehet egy használati eset vagy az alkalmazás tartományán kívül eső külső történés.

A példákban előforduló használati eseteket kiemeléssel hangsúlyozhatjuk. Ha lehetőségünk van rá, akkor a megfelelő belső hivatkozási rendszerrel olyan `hypertext` dokumentumot állíthatunk elő, ahol a kiemelt funkciókra kattintva megjelenik az azt tartalmazó használati eset diagram. Ugyancsak kiemelhetjük a külső felhasználókat, az aktorokat.

strukturálás

Az összetettebb résztevékenységeket, illetve a más feltételek mellett lezajló eltérő működéseket vagy a pontok strukturálásával szemléltetjük vagy külön alkalmazási példán vesszük fel. Strukturálás esetén azonban a túl sok szint megnehezíti a megrendelővel való egyeztetést. A cél mindig az áttekinthetőség megtartása.

Először célszerű a normális, a tipikus alkalmazásokat rögzíteni. Ezután adjuk meg a normálistól eltérő, speciális, például hibás esetek példáit.

Az alkalmazási példák egyben az alkotóelemek integrálását követő együttes tesztelés kiindulópontjai is.

passzív objektum változása

Az alkalmazási példa általában a valóság egy, de legfeljebb néhány entitásának, más néven „objektumának" a változásával kapcsolatos, amely az alkalmazás szempontjából passzív és többnyire csak a használati esetekként megfogalmazott tevékenységek hatására módosul.

Például egy kórház laboratóriumi rendszerének egy alkalmazási példája lehet a következő:

Beteg vizsgálata:

- A betegtől *mintát vesznek*
- A mintát *szétosztják* a vizsgálatokra
- A szétosztott mintákat különállóan vizsgálják:
  - Elvégzik és *rögzítik* a vizsgálatot
  - A *validálás* során az orvos ellenőrzi, hogy a mért érték megfelelhet-e a valóságnak
    - ➢ Érvénytelen érték esetén új vizsgálatot kell elvégezni, esetleg a betegtől új mintát kell venni.
  - ➢ A validált vizsgálatról *jelentés nyomtatható*
- Az összes vizsgálat befejeztével a beteget elbocsátják

A pontok elé helyezett ikonokkal jelölhetjük a tevékenységek jellegét, például

- Normál tevékenység vagy történés.
- ➢ Normáltól eltérő tevékenység, például kiterjesztés.
- \- Adott feltételtől függő változat, a működés lehetséges elágazása.

Forgatókönyvek

Az alkalmazási példákkal a használati esetekként megfogalmazott funkcióelemek időbeli egymásutániságát rögzítjük, azt a logikai összefüggést, amellyel megvalósítjuk az egyes felhasználói célokat.

A következő lépésben a követelményeket úgy pontosíthatjuk, ha most az egyes használati esetekre fordítjuk a figyelmünket, és azokat írjuk le részletesebben.

használati esetek részletezése

A részletezés egy egyszerű módja, ha minden használati esethez megadunk egy vagy több *forgatókönyvet* (`scenario`), amelyben felsoroljuk, hogy a funkció milyen, az aktor és az alkalmazás között lezajló párbeszédet igényel.

51. Ábra: Forgatókönyv

A `Szervező` *lemond jelentkezést*:
- A `Szervező` bejelentkezik
- Megjelenik a meghirdetett tanfolyamok listája
- Kiválasztva egy tanfolyamot megjelenik a jelentkezők listája
- Lemondja a jelentkezést

A forgatókönyvek megadásakor célszerű a párbeszédre, az interakciókra helyezni a hangsúlyt. A forgatókönyvek így olyan pontokból állnak, amelyek vagy a rendszer felé történő adatszolgáltatást, vagy a rendszer adatszolgáltatását, például eredményközlését írja le.

párbeszéd, interakció

Több, „oda-vissza" kommunikáció egyetlen lépésként is megadható. Ha szükséges, akkor az összevont lépéseket külön forgatókönyvben részletezhetjük.

A `centre-out` elv alapján először a normálnak, tipikusnak tekintett forgatókönyveket vegyük fel, majd ezután térjünk ki a speciális, például hibás esetek párbeszédére.

centre-out

52. Ábra: Forgatókönyv

A `Jelentkező` *előzetesen jelentkezik a tanfolyamra*:
- Megtekinti a meghirdetett tanfolyamokat
  - ➢ Megtekintheti egy-egy tanfolyam részletes tematikáját
  - ➢ Jelentkezik egy tanfolyamra
    - Megadja az adatait
    - ➢ Az érdeklődő félbeszakíthatja jelentkezését
    - Jelentkezik
    - A már jelentkezettek létszámától függően a rendszer
      - elfogadja a jelentkezést
      - módosítást ajánl fel

A használati esetek tekinthetők egy vagy több forgatókönyv címének, így a használati eset diagramok valójában a forgatókönyvek grafikus tartalomjegyzéke. Egy `hypertext` dokumentum esetén a használati esetre kattintva megjeleníthetjük a párbeszédet leíró forgatókönyveket.

Másik példaként tekintsük egy raktári rendszer vételezési használati esetének forgatókönyvét:

A `Raktáros` *vételez* a raktárból:

- A `Raktáros` jelzi, hogy vételezni kíván
- Megjelenik a raktározott áruk listája ábécé sorrendben. Minden tétel mellett az áruk együttes darabszáma is megjelenik.
- A `Raktáros` kiválasztja az árut
- A `Raktáros` megadja a vételezés mennyiségét.
  - ➢ „Nincs a raktárban a kért mennyiségű áru" hibát kapunk, ha nincs megfelelő mennyiségű áru.
- ➢ Ha a vételezett árufajta készlete a szükséges mennyiség alá csökkent, akkor ☝ *utánrendelés szükséges*.
- A rendszer jelzi, hogy a vételezés sikeres volt és erről nyomtat egy bizonylatot.

félig formalizált

A forgatókönyvek segítségével félig formalizált módon rögzíthetjük a felhasználóval történő egyeztetéseket (interjúkat). Azonban csak részben alkalmasak arra, hogy a megrendelők önállóan ellenőrizzék a forgatókönyvek helyességét.

Fontos, hogy a párbeszédek leírása során a felhasználó fogalmait használjuk és kerüljük az informatikai fogalmakat (`vételezés` panel, `Ok` gomb, stb.).

A forgatókönyvek hasonlítanak az alkalmazási példákra, az egyes pontokban azonban nem tevékenységeket, hanem interakciókat, az aktor és a rendszer közötti párbeszéd elemeit soroljuk fel.

Forgatókönyvek segítségével tisztázhatjuk az egyes használati esetek működését, így észrevehetünk olyan változatokat vagy részfunkciókat, amelyeket hangsúlyozni szeretnénk.

tesztforgatókönyvek

A forgatókönyvek egyben tesztforgatókönyvek is, így a modellezett vagy a már megvalósított alkalmazásról bebizonyítható, hogy az valóban a követelményeknek megfelelően működik.

eseményfolyam

Az UML a forgatókönyv mellett az *eseményfolyam* `(event-flow)` elnevezést is használja, melyet rövid mondatokból álló folyamatos szöveggel ad meg. A résztevékenységet `include (résztevékenység)`, a kiterjesztési pontot pedig `(kiterjesztési pont)` formában adhatjuk meg, a szövegen belül önálló mondatként.

A ***Q*** módszer a résztevékenység megadására egyszerű kiemelést ajánl, melyet kiterjesztési pont esetén a ☝ (szövegesen: ^) jel előz meg.

Működési leírások

A forgatókönyvek segítségével olyan használati eseteket írhatunk le, amelyekben a vezérlést alapvetően a felhasználói interakciók határozzák meg. Így nem alkalmasak olyan funkciók ábrázolására, amelyek viszonylag kevés interakcióra, ugyanakkor több és összetettebb tevékenységsorozatra épülnek.

A több, önálló lépésben lezajló működések a tevékenységsorozatok leírásával vázolhatók. A tevékenységsorozat egyben egy időbeli egymásutániságot is jelent, így megadhatjuk, hogy melyek az érvényes sorrendek.

használati esetek részletezése

A működési leírások különösen akkor hasznosak, ha a használati eset valami olyan folyamatot indít be, amely az aktortól részben függetlenül működik.

A működési leírás hasonlít az alkalmazási példára, így az egyes pontokban tevékenységeket (vagy más néven: műveleteket) adunk meg. Az összetett résztevékenységet vagy a más feltételek mellett lezajló eltérő működést strukturálással vagy külön működési leírással jelezhetjük. Először a normális működéseket adjuk meg, majd ezután a speciális, például a hibás eseteket.

tevékenység sorozat

Tanfolyam véglegesítése:

- Sorra veszi a tanfolyam nem lemondott jelentkezéseit
  - Ha a jelentkező közvetlenül értesíthető
    - Összeállítja a véglegesítés üzenetét
    - Az értesítés módjától függően
      - e-mail esetén elküldi a levelet
      - fax esetén feladja a fax-ot
- Összeállít egy dokumentumot a nem értesített jelentkezőkről

A leírás során célszerű a tevékenységeknek csak a vázolására szorítkozni. Minél kevesebb műveletet adjunk meg, kizárólag azokat, amelyek valóban lényegesek a felhasználó szempontjából. Fontos, hogy a felhasználó fogalmait használjuk és ne az informatikai nyelvezetet.

Az egyszerűbb tevékenységek a forgatókönyveken belül is megadhatók. A felhasználótól független, önálló működést azonban a legtöbbször célszerű kiemelni (például egy részfunkcióként) és külön megfogalmazni.

Felhasználói felületek

A forgatókönyvekkel, illetve a működési leírásokkal pontosíthatjuk az egyes használati esetek lépéseit, azaz a felhasználóval történő párbeszédet vagy a végrehajtandó tevékenységsorozatokat.

A forgatókönyvek egyes elemei az aktorként megjelenő felhasználóval vagy külső rendszerrel történő információátadást írnak le. A használati esetek pontosításának következő lépéseként célszerű megtervezni az információátadás eszközeit, a felhasználói felületeket.

információátadás eszközei

A *„felhasználói felület”* (`user interface`) kifejezést általános értelemben értsük, azaz a külső rendszerek is a rendszer felhasználói, melyekkel az adatcsere speciális módon, például adott szerkezetű fájlok felületén keresztül is történhet. Külső rendszerrel történő kommunikáció esetén megtehetjük, hogy az adatcserét vizualizáljuk úgy, hogy az adatforgalom elemeihez képernyőket és paneleket rendelünk.

hangsúly a kommunikáción

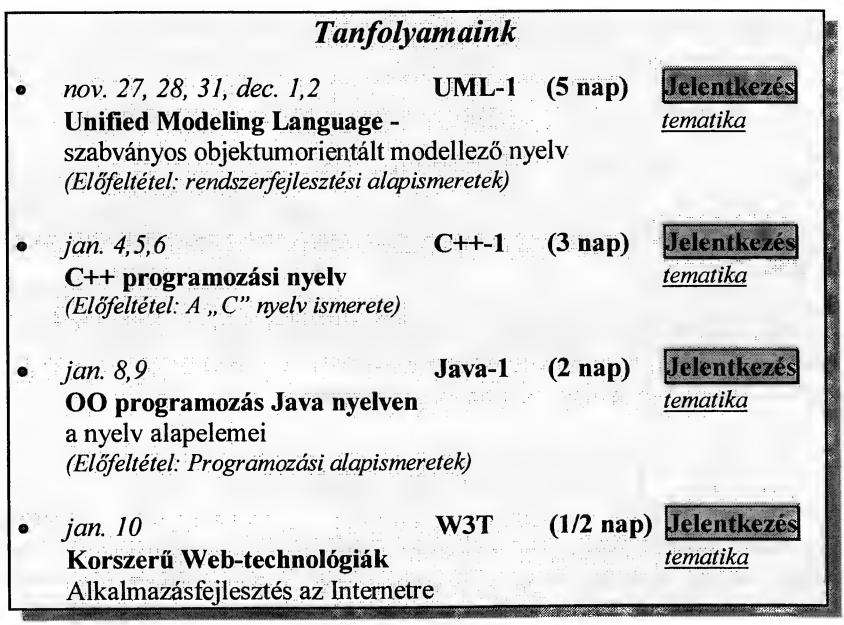
A felületek megadásakor nem szükséges a pontos és végleges formát megrajzolni, hanem a hangsúlyt a felhasználóval történő kommunikációra, azaz a bemenő és a szolgáltatott adatokra kell helyezni. Hasznos lehet, ha az első vázlatokat papíron készítjük el, hogy a használt eszköz ne kösse meg a kezünket és a fantáziánkat.

megrendelővel egyeztetés

A korábbi dokumentumokhoz hasonlóan a felhasználói felületek ugyancsak alkalmasak a megrendelővel történő egyeztetésekre. Segítségükkel ábrázolhatjuk, hogy a rendszer számára milyen adatok adhatók meg, illetve abból milyen adatok nyerhetők ki. Ugyanakkor maga a felhasználó is jelezheti, hogy bizonyos tevékenységekhez még további adatok is szükségesek. A felhasználói felületek tekinthetők az alkalmazásban elrejtett belső szerkezet külső képének, így a felületekkel ábrázolhatók az alkalmazáson belül tárolt adatok. A felhasználói felületeken történő változtatás pedig visszavetíthető a rendszerbe.

belső modell kivetített képe

53. Ábra: Előzetes felhasználói felület



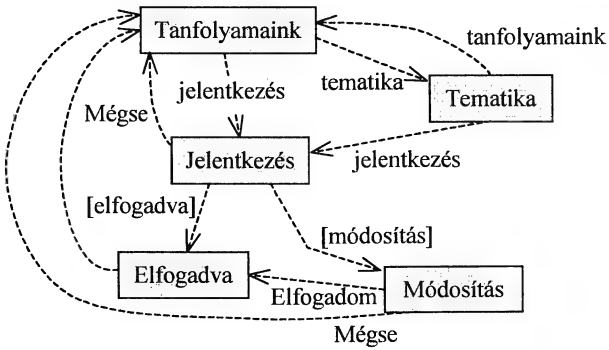
A forgatókönyvek lépései általában egy-egy felhasználói felületen lezajló interakciót írnak le. A forgatókönyveken belül kiemelhetjük ezeket a hivatkozásokat,

illetve `hypertext` dokumentum esetén hozzákapcsolhatjuk a felületek képét és leírását.

A felhasználói felületek közötti kapcsolat, azok egymásutánisága ábrázolható egy olyan diagramon, ahol az egyes paneleket téglalapokként jelöljük, melyeket szaggatott vonalú nyilakkal kötünk össze. Az adott panelből kiinduló vonalon megadható, hogy milyen esemény, illetve feltétel hatására jelenik meg az újabb panel (pl.: „`Ok`"). A feltételeket az esemény mögött, szögletes-zárójelek között adjuk meg (pl.: „`[hibás jelszó]`" vagy „`Postázás[hiányzó cím]`").

áttekintő diagram

54. Ábra: Felhasználói felületek kapcsolata



Áttekintésül készíthetünk olyan nagyméretű diagramokat, ahol a paneleket nem alakzatként, hanem konkrét megjelenési formájukban ábrázoljuk.

A forgatókönyvek alaplépései a panelek közötti navigációt írják le, így a diagramon egy bejárási utat határoznak meg. A forgatókönyvek megadásakor az első lépéseket célszerű nem a „kályhától" (bejelentkezéstől, főmenütől, stb.) kezdeni. Kiindulási pontként válasszunk ki egy panelt, amelyet a forgatókönyv első sorában megadunk.

A felhasználói felületek szoros kapcsolatban vannak a használati esetekkel. A használati esetek az aktor és az alkalmazás közötti párbeszédet írják le, amelyek a felhasználói felületeken keresztül zajlanak le. A használati esetek ezért tekinthetők egyfajta absztrakt felhasználói felületként, az aktor absztrakt lehetőségeiként.

felhasználói felületek és használati esetek

A felhasználói felületeket, illetve azoknak az információt kérő, közlő és vezérlő elemeit egyaránt modellelemeknek tekintjük, így azokat is lássuk el rövid, egy-két soros magyarázattal, értelmezéssel.

A megrendelőkkel való egyeztetést megkönnyíthetjük, ha elkészítjük a felhasználói felületek prototípusát, egy olyan működő „programot", amely képes tesztadatokkal megjeleníteni és egyszerű vezérlésekkel összekapcsolni a paneleket.

felhasználói felületek prototípusa

Előzetes felhasználói leírás

„Utolsó" szöveges dokumentumként készítsük el a felhasználói leírás vázlatát. Az „utolsó" jelző azért került idézőjelek közé, mert ebben foglaljuk össze az eddig összegyűjtött követelményeket.

felhasználó lehetőségeinek körvonalazása

A vázlat készítése során célszerű a felhasználó lehetőségeinek körvonalazására, és nem a részletességre törekedni. Szükségtelen például a menük és a felhasználói felületek pontos megjelenési formáját megadni, kiosztani a gyorsítóbillentyűket, vagy meghatározni az adatbekérések sorrendjét.

alkalmazás lehetőségei és céljai

Az előzetes felhasználói leírás célja, hogy a megrendelő áttekinthesse a készítendő alkalmazás lehetőségeit. Megismerje, hogy mely céljait és milyen módon valósítja meg a rendszer, illetve az milyen adatokat fog kérni, szolgáltatni és tárolni.

Az előzetes felhasználói kézikönyvet a következő alapszerkezetre építhetjük rá:

- *Áttekintés:* összefoglalja az alkalmazás alapvető célját és a szituációt.
- *Áttekintő csomag és használati eset diagram:* a legfontosabb részek és funkciók grafikus tartalomjegyzéke. A diagramot követően a fogalomszótárból felsoroljuk a modellelemek elnevezését értelmező részeket.
- *Szemléltetések:* itt különböző magyarázatokat és szemléltető diagramokat helyezhetünk el.
- *Alkalmazási példák:* megmutatják, hogy az egyes felhasználói célok mely használati esetekkel valósíthatók meg.
- *Csomagok alapján részekre bontjuk a kézikönyvet:* minden rész ugyancsak a fenti szerkezettel (áttekintéssel, stb.) kezdődik.
- *Felhasználói felületek diagramja:* a felhasználói felületek grafikus tartalomjegyzéke.
- *Felhasználói felületek:* az egyes fejezetek hierarchikus beosztása a felhasználói felületek szerkezetét követi. A felületek megadásakor az információcserére és nem a pontos megjelenési formára helyezzük a hangsúlyt.
- *Forgatókönyvek és működési leírások:* adott felhasználói felület fejezetén belül adjuk meg a külön nem részletezett párbeszédeket, illetve a beindítható tevékenységsorozatokat.

Az alapszerkezetet szabadon bővíthetjük a szükséges kiegészítésekkel. Például gyakori, hogy a szemléltetésekben a szituációt ábrázoló további diagramokat adunk meg.

Követelményelemzés

A Q modellező módszer a fejlesztés első lépéseként a követelmények elemzését ajánlja, mely az alkalmazásnak a felhasználó által látható képét rögzíti. A dokumentumok felhasználhatók a megrendelőkkel történő egyeztetésre, segítségükkel megtervezhető a fejlesztés menete, meghatározhatók a kritikus pontok, megbecsülhető a szükséges erőforrásigény és ezek a későbbi tesztek kiindulópontjai is.

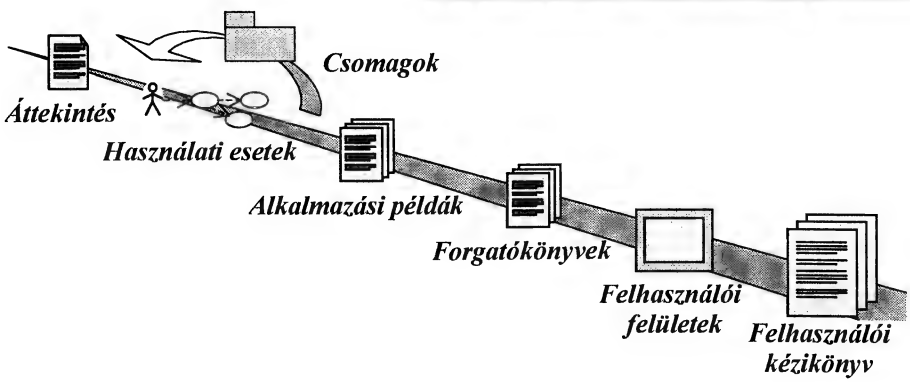
A fejlesztés első lépéseként célszerű összegyűjteni a készítendő alkalmazással szemben támasztott követelményeket. Célunk, hogy valóban az igényelt alkalmazást fejlesszük ki, ne pedig azt, amelyet mi szeretnénk megvalósítani. A siker érdekében meg kell becsülnünk a fejlesztés erőforrásigényét, és fel kell derítenünk a kritikus pontokat.

félig formalizált szöveges dokumentumok

A *Q* módszer a követelmények elemzésére félig formalizált, de alapvetően szöveges dokumentumokat ajánl. Ha alaposabban megnézzük ezeket a szövegeket, akkor észrevehetjük, hogy az objektumorientáltságra, annak szemléletére és fogalmaira egyetlen szó sem utal. Ez az egyik legnagyobb előnyük, mivel a felhasználót nem kényszerítik egy tőle idegen fogalomrendszer elsajátítására.

Valóban igaz, hogy az igényeket rögzítő dokumentumok nem OO szemléletűek. A használati esetek, illetve a köréjük csoportosuló dokumentumok egy másik szemléletet képviselnek, amelyre a *Q* módszer is épül. Ebben a megközelítésben a követelmények rögzítése során a fekete dobozként tekintett alkalmazás külső képét, külső kapcsolódási pontjait (a használati eseteket) és a kapcsolat adatcseréjének protokollját (a felhasználói felületeket) adjuk meg. A *Q* módszer újdonsága, hogy ezt a szemléletet következetesen és teljes spektrumban alkalmazza mind az igények rögzítésére, mind a fejlesztés későbbi szakaszaiban.

55. Ábra: A követelményelemzés vázlata



felhasználó fogalmai

A követelményelemzés során előállított dokumentumokban a felhasználó nyelvezetét használjuk és kerüljük a megvalósítással kapcsolatos technikai és informatikai fogalmakat. Az igényeket a felhasználóval történő megbeszélések (interjúk) és folytonos egyeztetések során gyűjtjük össze, így biztosíthatjuk, hogy az nem lesz idegen a számára és nem az elképzeléseitől eltérő terveket készítünk, kárbeveszett munkával.

A dokumentumok alapvetően szöveges jellegűek, így értelmezésük a felhasználótól nem igényel különösebb előképzettséget. Ugyanakkor az alapszerkezet és a pontokban történő felsorolás biztosítja, hogy valóban összegyűjtsük a szükséges információkat és ne csak bizonytalan körvonalú és félreérthető víziókkal kíséreljük meg az egyeztetést. A dokumentumokkal a felhasználóktól megszerzett információk félig formalizált módon rögzíthetők.

Kidolgozás

A dokumentumokat nem célszerű egymás után, a leírt sorrendnek megfelelően teljes részletességgel kidolgozni. Ehelyett minden dokumentumot egyszerre bővítsünk és pontosítsunk a „centre-out" technikának megfelelően, azaz először a hangsúlyosabb, majd ezután a kevésbé lényeges területekre fordítsuk a figyelmünket, például először a gyakori normál eseteket derítsük fel, majd a speciális és hibás változatokat.

centre-out

Például kezdjük a követelményelemzést egy áttekintő leírással, majd rajzoljunk egy vázlatos használati eset diagramot. Közben készítsünk alkalmazási példákat, melyekkel szemléltethetjük, hogy a funkciók milyen módon teszik lehetővé a felhasználói célokat. Ezután a használati esetekhez adjunk meg forgatókönyveket, így felderíthetjük az említést érdemlő változatokat, illetve részfunkciókat. A felhasználókkal készített interjúkat fogalmazzuk meg forgatókönyvekben, melyeket egy rövid elnevezéssel vegyünk fel új használati esetekként.

A lényeges használati esetekből, illetve forgatókönyvekből kiindulva rajzoljuk meg néhány felhasználói felület tervét. A felhasználói felületeket egyeztessük a megrendelőkkel, illetve segítségükkel rendezzük át és pontosítsuk a diagramjainkat és a forgatókönyveket.

Ha észreveszünk összetartozó funkciócsoportokat, akkor az alkalmazás használati eseteit bontsuk csomagokra. Minden modulra (csomagra) készítsünk önálló áttekintést, részletesebb használati eset diagramokat, alkalmazási példákat és forgatókönyveket. A forgatókönyvek olyan részletesek legyenek, hogy segítségükkel a megvalósított alkalmazás helyes működése ellenőrizhető legyen. A felhasználói interfészeket rendeljük az egyes csomagokhoz.

Az elkészített dokumentumokat rendezzük egy előzetes felhasználói kézikönyv formájába. Ügyeljünk arra, hogy alapvetően a szakterület és a felhasználók fogalmait használjuk és kerüljük a megvalósítással kapcsolatos technikai részleteket, informatikai fogalmakat. Az előzetes kézikönyv mellé célszerű elkészíteni a felhasználói felületeket megjelenítő prototípust, amely a szöveges tervek mellett kézzelfogható és kipróbálható képet ad az alkalmazásról.

A követelményelemzés célja

felhasználói kézikönyv

A követelmények összegyűjtése során valójában egy adott alapszerkezetű felhasználói kézikönyvet, valamint az alkalmazásnak a felhasználói felületeket megjelenítő prototípusát állítjuk elő. Ezzel egyetlen sort sem írunk le feleslegesen, mivel a későbbiekben az előzetes kézikönyvet a végleges változatban, a prototípust pedig magában az alkalmazásban használjuk majd fel. Nem rögzítünk felesleges vagy gyengén specifikált követelményeket, és nem készítünk a kelleténél korábban programkódot, amit a későbbiekben valószínűleg úgy is meg kellene változtatni. Ugyanakkor összegyűjtjük és egy áttekinthető formába rendezzük az egyeztetéshez és a fejlesztés következő lépéseihez szükséges igényeket.

A követelmények tisztázása egyszerre több célt is szolgál:

- Megrajzoljuk a *rendszer körvonalait*, határait, így a megrendelővel történő egyeztetések eredményeként véglegesítjük, hogy mi kerüljön be, és mi maradjon ki a rendszerből.
- Meghatározzuk az alkalmazás *alapvető részeit és funkcióit*, így több lehetőségünk van megbecsülni a fejlesztés *erőforrásigényét*.
- Fel tudjuk deríteni a kérdéses és a *kockázatos pontokat*.
- Meg tudjuk határozni, hogy az egyes elemek megvalósítása esetén milyen sorrendet kövessünk, azaz meg tudjuk tervezni magát a *fejlesztés menetét*.
- A fejlesztés *következő lépéseihez* elegendő információt gyűjtünk össze.
- Az alkalmazás kívülről látható képe egyben a *tesztek* kiindulópontja is.

iterációk meghatározása

Az alapvető használati esetek összegyűjtése után célszerű meghatározni a kidolgozás sorrendiségét, hogy mely funkciókat kell korábban megvalósítani, mivel alapvető szerepűek a rendszer működése szempontjából és melyek hagyhatók a fejlesztés későbbi fázisaira. A sorrendiséget iterációs ciklusokként fogalmazzuk meg, melyek mind működő rendszereket, a teljes alkalmazás kipróbálható részeit állítják elő. Az iterációkat megelőzhetik a teljes alkalmazásra vonatkozó igények vázlatos felmérése, melyeket (a `Rational Unified Process` alapján) *előzetes iterációknak* (`preliminary iteration`) nevezünk. Alapvető fontosságú, hogy minden iterációnál konkrét célokat, tevékenységeket és határidőket határozzunk meg. Optimális esetben a megrendelő is partner az iteratív fejlesztésben. Ez számára is előnyös, mivel nyomon követheti, és ellenőrizheti a fejlesztést, folyamatosan veszi át és tesztelheti az alkalmazást, sőt, a költségek is több, kisebb részletben oszlanak meg, melyek előre tervezhetők.

szakterületi elemzés

A követelmények összegyűjtése után a következő lépés a szakterületi elemzés, amely már nem az igények, hanem a megvalósítás oldaláról közelít. A követelményelemzés utolsó lépéseit célszerű összekapcsolni a szakterületi modell felépítésével, mivel így ellenőrizhetjük a megvalósíthatóságot és a helyességet. A teljes elemzés (a felületek prototípusával együtt) időben és költségekben körülbelül a fejlesztés ötödehatoda, így ha lehetséges, akkor az külön szerződés keretében is elkészíthető. Ha az alkalmazásra már előre megszabott ellenérték áll rendelkezésre, akkor fordítva, a funkciókat méretezhetjük a költségkeretekhez.

II.

Osztálydiagramok

Objektumorientált modellek esetén alapvető jelentőségűek az alkalmazás strukturális szerkezetét leíró osztálydiagramok, melyeken osztályokat, valamint a közöttük lévő viszonyokat ábrázolhatunk.

Az osztályokat az attribútumaikkal és műveleteikkel, azaz a jellegzetességeikkel adjuk meg. Összetett műveleteket a „felelősségek megosztásával", az objektumok együttműködésével valósítjuk meg. Az együttműködő objektumok a közöttük vezető kommunikációs útvonalak, más néven kapcsolatok segítségével érik el egymást. A több osztályban ismétlődő jellegzetességek egy általánosabb típusként megjelenő ősosztályba emelhetők ki. Az ősosztály a valós világban egy általánosabb fogalomnak felel meg, mellyel több változat közös jellegére hivatkozhatunk.

Osztály

Az objektumorientált szemlélet a valóság fogalmait osztályokként modellezi, melyeket attribútumaikkal és műveleteikkel adunk meg. Az attribútum az objektumot adott szempontból tekintve egy érték-jellegű adattal jellemzi. Az objektum összetett viselkedését műveletekre bonthatjuk fel. A művelet egy olyan kérdés, amelyre az objektum válaszolni tud, vagy egy olyan transzformáció, amely megváltoztatja az objektum jellemzőit.

Az alkalmazással szemben támasztott követelmények tisztázásakor alapvető jelentőségűek a használati eset diagramok, melyek az alkalmazás kívülről látható képét grafikus módon, térképként jelenítik meg. A használati esetek köré szervezzük az igényeket leíró többi dokumentumot, például a forgatókönyveket és a működési leírásokat. Az alkalmazás belső szerkezetének ábrázolása szempontjából hasonló szerepűek az alkalmazás fogalmait és azok viszonyait leíró osztálydiagramok, melyek így nem a felhasználó, hanem a fejlesztő szemszögéből alapvető jelentőségűek. Osztálydiagramokkal ábrázoljuk az alkalmazás belső felépítését, az architektúrát és azok köré szervezünk több diagramtípust, melyekkel a nem strukturális (hanem például az időbeli) viszonyokat szemléltetjük.

alkalmazás belső felépítése

Objektum és osztály

Az alkalmazás belső szerkezete a valóságnak a szakterület által lefedett részletén alapszik. A valóságnak ezt a részletét és egyben az alkalmazás alapszerkezetét úgy írhatjuk le, ha megkeressük és jellemezzük annak építőelemeit, valamint a közöttük lévő viszonyokat.

A valóság alapelemeit, szerkezetének „csomópontjait" *entitásoknak* vagy *egyedeknek* `(entity)` nevezzük, amivel arra utalunk, hogy a rendszer többé-kevésbé különálló, függetlenül vizsgálható és egymástól megkülönböztethető részei. Az egyediség egyfajta „körülrajzolhatóságot" jelent, azt, hogy el tudjuk választani a rendszernek az entitáshoz tartozó részét annak környezetétől. Az entitás lehet a valóság egy konkrét, megfogható része, de lehet egy fogalom is.

entitás

Az objektumorientált szemlélet szóhasználatában az entitásokat *objektumoknak* `(object)`, vagy más néven *példányoknak* `(instance)` nevezzük. Az objektumokat általában nem egyenként és külön definiáljuk. Ehelyett az azonos jellegzetességű példányoknak a *típusát* adjuk meg, amelyet *osztálynak* `(class)` nevezünk. Az osztály az azonos jellegzetességű, más szóval egy típusba tartozó objektumokat csoportosítja, azaz egy absztrakt objektumot határoz meg.

objektumok, példányok

osztály

Az azonos jellegzetességű objektumok közös osztályba sorolását *osztályozásnak* `(classification)` nevezzük. Az osztály egy általános keretet határoz meg, amelyet az objektumok töltenek fel konkrét tartalommal.

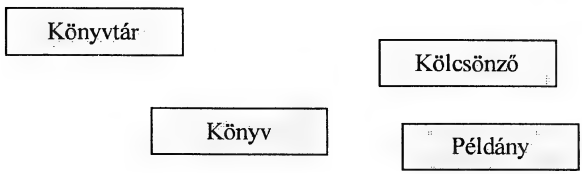
osztályozás

Régebben, a '90-es évek elején az objektum szó kettős jelentésű volt, így jelenthetett konkrét *(objektum példányt)*, vagy absztrakt objektumot *(objektum osztályt)* is. Napjainkban az objektum szót már „példány" értelemben használják, míg az absztrakt objektumokat az „objektum osztály" szókapcsolat helyett röviden osztálynak nevezzük.

jelölés

Az UML jelöléseivel az osztályokat egy téglalappal („dobozzal") jelöljük. A téglalap felső részében középre igazítva adjuk meg az osztály nevét. Az UML ajánlása szerint elnevezésként egyesszámú főnevet vagy rövid főnévi szerkezetet használjunk, melyet nagybetűvel kezdünk.

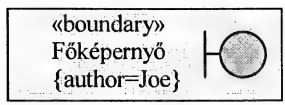
56. Ábra: Osztály



névrész

A modellelemek alakzatának felső része, ahol az elnevezés is szerepel, az UML szóhasználatában a *névrész* `(name compartment)`. Mint minden modellelem esetén, az osztályok névrészében is megadhatunk sztereotípiát (az elnevezés felett) és megszorítást (a név alatt).

57. Ábra: Sztereotípia és megszorítás



Modellezett rendszerünket a későbbiekben át szeretnénk képezni az informatika világába. Ez azt igényli, hogy minden egyes koncepciónak a számítástechnikai megfelelőjét is meg kell határoznunk.

Az objektumok a jellemzőiket leíró és tároló adatcsomagokként jelennek majd meg, például egy programnyelv adatstruktúrájaként. Relációs adatbázisban (RDBMS) történő tárolás esetén az osztályt egy adatbázistáblaként képezzük le, az osztály objektumait pedig rekordokként ábrázoljuk.

objektum-azonosító

Az objektumok egyedisége azt jelenti, hogy az objektumot valamilyen módon azonosítani tudjuk, így azt egyértelműen megkülönböztethetjük más objektumoktól. Az objektumorientált környezetnek így szolgáltatnia kell egy *objektumazonosítót* `(OID: object identifier)`, amelytől két dolgot követelünk meg:

- legyen *egységes* formátumú, így segítségével a rendszer bármely objektumára hivatkozhassunk, és
- *egyértelmű* legyen, azaz pontosan egy objektumot határozzon meg.

A szakterületi modell első elemei

A „honnan jönnek az osztályok" az objektumorientált módszerek alapvető kérdése. A `Rational Unified Process` egyszerűen az alkalmazás szakterületével kapcsolatos fogalmak összegyűjtését és osztályként történő felvételét ajánlja. A szöveges megfogalmazáson alapuló megközelítések — mint az OMT módszere is — azt javasolják, hogy a feladat szövegéből keressük ki és húzzuk alá a főneveket, amelyek lehetséges osztályokra utalhatnak. Sajnos mindkét technika nagyon sok felesleges osztályt vesz fel és nem ad választ például olyan kérdésekre, hogy mely aktoroknak szükséges bekerülni az alkalmazásba és melyeknek nem.

felhasználói felületek alapján

A ***Q*** módszer javaslata szerint az alapvető osztályokat a külvilággal történő adatcsere protokollja, azaz a felhasználói felületek alapján keressük meg. A felületek azok az eszközök, amelyekkel a valóság megfelelő részletét az alkalmazás belső rendszerébe képezzük át, illetve a belső képet azon keresztül jeleníthetjük meg.

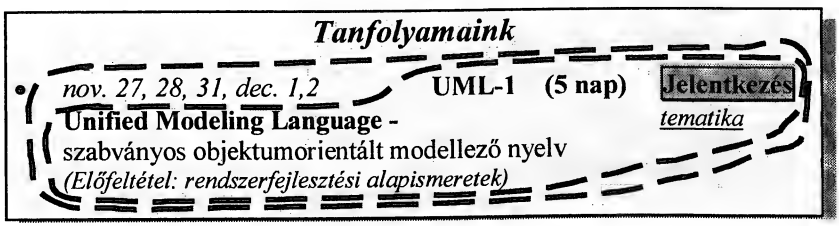
A felületek a külső entitásokra vonatkozó adatcsere eszközei is, így nincs más dolgunk, mint az összes felületen körberajzolni az összetartozó, egy entitásra vonatkozó adatok csoportját, megkeresni a megfelelő szakterületi fogalom nevét és azt felvenni osztályként. Az első vizsgálataink során hagyjuk ki a felületek vezérlésre szolgáló elemeit.

58. Ábra: Entitás adatainak körberajzolása

„invariancia-elv"

Az adatok összetartozását egy „invariancia-elvnek" nevezhető technikával kereshetjük meg, mely szerint vizsgáljuk meg, hogy mely adatok értékkombinációja ismétlődhet úgy, hogy mellette más adatok változnak. Példánkban az azonos témájú, de más időpontokban megtartott tanfolyamok esetén csak néhány adat, például az időpontok változnak. Így megtalálhatunk egy újabb entitást, amelyet például `Tanfolyam típusnak` nevezhetünk el.

59. Ábra: Ismétlődő értékkombinációk



Példánkban a felhasználói felületeken az aktorok közül csak a `Jelentkező` adatai jelennek meg, így csak az kerül be a modellünkbe.

Az azonosított entitásokat vegyük fel egy osztálydiagramon. A fogalomszótárban minden osztály rendszerben betöltött szerepét foglaljuk össze egy-két bekezdésben.

60. Ábra: Osztályok

Attribútumok

Az entitások jellemzésének egyik módja az attribútumainak felsorolása. Az *attribútum* `(attribute)` az entitás adott szempontból vett tulajdonságát írja le.

attribútum

Az UML jelöléseivel az osztály attribútumait a névrész alatt, attól egy vonallal elválasztva adhatjuk meg a következő formában:

jelölés

```
név: típus = kezdeti érték
```

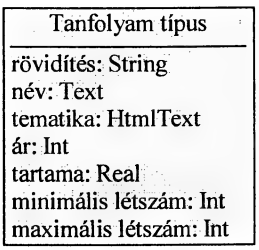
A megadási módot a következőképpen értelmezhetjük:

- az attribútum-név elnevezi a szempontot,
- egy objektum konkrét attribútum-értéke megadja, hogy az objektum milyen az adott szempontból,
- az attribútum típusa meghatározza a lehetséges értékeket,
- az opcionális kezdeti érték az objektum készítésekor beállítandó értéket jelenti.

Az attribútum neve előtt sztereotípia, a definíció végén pedig megszorítás is megadható. Attribútum-névként az UML kisbetűvel kezdődő egyes számú főnevet vagy főnévi szerkezetet ajánl.

Az attribútum típusát a modellünk első vázolásakor nem szükséges azonnal meghatározni.

61. Ábra: Attribútumok



adat és érték

Az osztály attribútumai olyan mintát, „űrlapot" definiálnak, amelyeket a konkrét objektumok tölthetnek fel különböző vagy éppen azonos értékekkel. Önmagában az érték csak egy csupasz adat (például: 5), amelyet az attribútum nevével együtt értelmezhetünk információként (az OOAD-1 tanfolyam időtartama 5 nap).

Az objektumok egyedi azonosítóját (OID) nem szabad külön attribútumként felvenni, mivel úgy tekintjük, hogy azt az objektumorientált környezet szolgáltatja. Minden objektum esetén feltételezhetjük, hogy az rendelkezik egyfajta egyedi azonosítóval, ami lehet például a rá mutató pointer, vagy relációs adatbázis táblájában történő tároláskor egy automatikusan képzett (pl. `tanfolyamTípusID` nevű) egyedi kulcs.

Műveletek

viselkedés

Az entitások jellemzésének egy másik módja, ha megadjuk azok *időbeli változását*, vagy az OO zsargont használva: *viselkedését* `(behavior)`. Az összetett változást az egyszerűbb kezelhetőség érdekében kisebb darabokra, úgynevezett *műveletekre* `(operation)` tördelve írjuk le.

A művelet egyszerűen egy olyan eljárás, amelyet az objektum végre tud hajtani vagy olyan kérdés, amelyre az tud válaszolni.

szignatúra

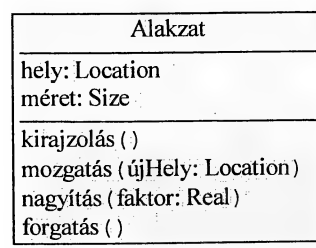
A műveletet a nevével és a paramétereinek típusával azonosítunk, amelyet együttesen a művelet jelének, specifikációjának vagy *szignatúrájának* `(signature)` nevezünk. Mivel az azonosítás a paraméterek típusaival együtt történik, ezért több műveletnek is lehet azonos a neve, ha a paramétereik különböznek. Ezt a lehetőséget a műveletek *„túlterhelésének"* `(overloading)` nevezzük.

túlterhelés

metódus

Minden objektum esetén a szignatúrával azonosított művelethez tartozik egy *metódusnak* `(method)` nevezett utasítássorozat, amely a műveletet adott módon valósítja meg. A metódus az absztraktnak tekinthető művelet konkretizálása, ezért itt is típus/példány kapcsolat van, hasonlóan az osztály és annak objektumainak viszonyához.

62. Ábra: Műveletek



Az UML a műveleteket az attribútumok alatt sorolja fel, azoktól egy újabb vonallal elválasztva.

Egy műveletet a következő módon adhatunk meg: *jelölés*

```
név(paraméterek):típus
```

- A név a művelet neve, amely „túlterhelhető".
- Az opcionálisan megadható típus a visszatérési érték típusa.

A paramétereket vesszővel elválasztva a következő formában soroljuk fel:

```
jelleg név: típus = alapértelmezett érték
```

- a *jelleg* az `in`, `out` vagy `inout` kulcsszó egyike, melyek közül az alapértelmezett az `in`.
  - `in`: a paraméternek felhasználhatjuk az értékét, de annak változtatása nem fog kihatni a műveletet hívóra,
  - `out`: a paraméteren keresztül értéket adhatunk vissza,
  - `inout`: a paraméter értékét felhasználhatjuk és azon keresztül értéket is utalhatunk vissza
- a *név* a paraméter neve,
- a *típus* a paraméter által felvehető értékeket adja meg,
- az *alapértelmezett érték* opcionális.

A művelet végrehajtásának részleteit a művelethez kapcsolt megjegyzésen határozhatjuk meg. Ha a műveletet egy adott programozási nyelven írjuk le, akkor azt kapcsoszárójelek közé kell zárni.

Az attribútumokhoz hasonlóan a név előtt sztereotípia, a definíció végén pedig megszorítás is szerepelhet.

Az objektumorientált szemlélet egyik újdonsága, hogy az összetartozó adatcsoportokhoz megadhatjuk az azzal kapcsolatos, azokat kezelő műveleteket. Az adatok manipulációi így nem a programban szétszórva helyezkednek el, hanem azokat kigyűjthetjük, így rövidebb, áttekinthetőbb és könnyebben módosítható alkalmazást kapunk.

Lekérdezés és módosító

A műveleteket alapvetően két csoportba soroljuk attól függően, hogy a végrehajtást követően észlelhető-e változás az objektumon.

A *lekérdezés* `(query)` mindössze egy kérdést fogalmaz meg, értékeket kér le az objektumtól anélkül, hogy kívülről észlelhető változást eredményezne. Ezzel szemben a transzformáció vagy *módosító* `(modifier)` a nevének megfelelően módosítja az objektum megfigyelhető állapotát. *lekérdezés* *módosító*

A *megfigyelhető állapot* `(observable state)` az objektum külső képét jelenti. Bizonyos megvalósítási technikák miatt a lekérdezés módosíthat a *tényleges* (de nem a megfigyelhető) állapoton, például egy hosszabb időtartamú számítás eredményét *gyorsítótárként* `(cache)` szolgáló attribútumban tárolhatja. A külső kép azonban ekkor is változatlan marad. *megfigyelhető állapot*

A lekérdezések jelölésére az UML az `{isQuery}` megszorítást javasolja. *{isQuery}*

Származtatott attribútum és osztály

származtatott és alapelem

Származtatottnak nevezzük a modell azon *elemét* `(derived element)`, amely valamely más elemekből számítható, más néven: származtatható. A nem származtatott elem az *alapelem* `(base element)`.

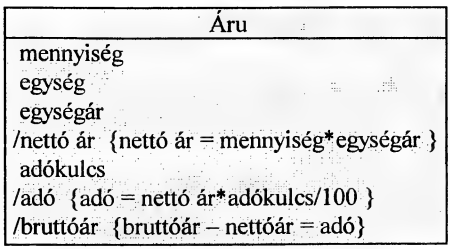
A származtatott jelleget a név elé vagy az alakzatra írt „/" jelöli.

származtatott attribútum

A *származtatott attribútum* `(derived attribute)` értéke más (alap- vagy származtatott) modellelemekből számítható, például egy személy születési dátumából számítható az életkora. Hasonlóan, a *származtatott osztály* (`derived class`) objektumait más elemekből állíthatjuk elő.

A származtatás módját megszorításként adhatjuk meg, melyet elhelyezhetünk az elem mögött, az alakzat mellett, vagy egy ahhoz csatolt külön megjegyzéslapon.

63. Ábra: Származtatott attribútumok



Fowler és Scott ajánlása szerint a származtatást az elemzés kezdetén az elemek közötti viszonnyal célszerű megadni (például egyenlettel: `bruttó ár-nettó ár = adó`), nem pedig egy művelet eredményeként. Az összefüggésből később kiemeljük a számított elemet.

A származtatás egy olyan speciális, függvényjellegű megszorítás, amely az alapul vett értékekből más értéket vagy értékeket egzaktul meghatároz.

A származtatott attribútum a programban olyan lekérdező műveletként jelenik meg, amelyeknek nincs paramétere. Tehát egy olyan kérdést jelöl, amelyre az objektum válaszolni tud.

tárolt származtatott attribútum

Bizonyos származtatott attribútumok, illetve lekérdezések eredményeit csak hosszabb időtartamú számítással kaphatjuk meg. A fejlesztés későbbi fázisában, a tervezés során dönthetünk úgy, hogy ezek eredményeit *gyorsítótárakban* `(cache)` tároljuk.

Láthatóság

bezárás

Az objektumorientált jelölések általában lehetőséget adnak arra, hogy az objektumnak meghatározzuk a külvilág számára látható részét és elrejtsük annak saját attribútumait és műveleteit. A *bezárásnak* `(encapsulation)` nevezett technikával az objektumot egy „fekete dobozba" zárva elrejthetjük a megvalósítás részleteit, így a rendszerünk áttekinthetőbb és könnyebben módosítható lesz.

Az UML a *láthatóság* `(visibility)` három szintjét definiálja:

- `+ (public)` jelöli a *publikus* (más néven: *nyilvános*), mindenki által látható modellelemet. *public*
- `# (protected)` jelöli a *védett* elemeket, melyek az objektum határain kívül nem láthatók. *protected*
- `- (private)` jelöli a *saját* (más néven: *privát*) elemeket. *private*

A védett és a saját láthatóság csak pontosítás esetén különbözik egymástól. Az osztály pontosított változata még a „határokon belül" van, ezért látja az általánosabb osztály védett elemeit, de nem használhatja azokat, amelyeket az általános a sajátjaként jelölt meg. Az osztályok általánosítását és pontosítását a későbbiekben részletesen tárgyaljuk.

A láthatóság jelét a modellelemek elnevezése elé (de a sztereotípia után) írjuk. Az attribútumok és műveletek mellett az osztályoknak és a csomagoknak is jelölhető a láthatósága.

A láthatóság megszorításként is megadható (például `{public}`).

A programozási nyelvek a láthatóság esetén általában az UML jelöléseivel azonos szinteket határoznak meg. Jegyezzük meg, hogy a Java rendelkezik egy alapértelmezett (jelölés nélküli) „csomag" `(package)` láthatósági szinttel, amely azt jelenti, hogy az osztályt, attribútumot vagy műveletet a csomagon belül lévő más elemek is elérhetik, de a csomagon kívül az rejtve marad. *„csomag" láthatóság*

Lekérdező és beállító műveletek

Az OO programozásban gyakran használatosak a lekérdező és a beállító metódusok. A *lekérdező metódus* `(getting method)` visszatérési értékként egyszerűen egy attribútum értékét utalja vissza, a *beállító metódus* `(setting method)` a paraméterének értékét pedig az attribútumba tölti át. *lekérdező és beállító metódus*

A technika segítségével az attribútumot elrejthetjük és a használatát csak adott műveleteken keresztül engedélyezzük. Így kontrollálhatjuk az attribútum elérését, például késleltetjük a kiszámítását, vagy beállítása esetén további tevékenységeket is végrehajthatunk. Gyakori felhasználási mód, hogy csak az attribútum olvasását engedélyezzük, ezért csak a lekérdező metódust adjuk meg.

Az azonosíthatóság érdekében a két művelet elnevezését az attribútum nevéből (mint alapnévből) egy prefix segítségével képezzük. A prefix lehet például lekérdezésnél `get` vagy logikai érték esetén `is`, míg beállításnál `set`. Például a következő

```
-address: String
```

„saját" attribútum esetén a publikus lekérdező és beállító műveletek a következő szignatúrájúak:

```
+getAddress(): String
+setAddress(newAddress: String)
```

Ha kívülről csak az attribútum olvasását szeretnénk engedélyezni, akkor nem adunk meg beállító műveletet.

Attribútumok és műveletek értelmezése

A *Q* módszer eltér az objektumorientált paradigma egyes fogalmainak hagyományos értelmezésétől. Szemlélete nem az objektumként modellezett entitás belső tartalmára helyezi a hangsúlyt, hanem annak környezetével vett kapcsolataira. Az entitásokat így „funkcionális csomópontoknak" tekinti, ahol a kívülről látható funkciók a lehetséges kapcsolatokat jelölik.

attribútumok értelmezése

Ebben a megközelítésben az attribútumok nem csak a közvetlen adattárolás eszközei, hanem az entitás olyan jellemzői, amelyek lekérdezhetők és esetleg beállíthatók. Mivel az objektumtól általában adatokat kérünk le, ezért az attribútumokat alapértelmezésben az objektum határain kívül csak olvashatónak tekintjük. A módszer ezzel eltér az UML szabványos értelmezésétől, de így kisebb méretű és kezelhetőbb modellt kapunk, az eltérések és a kiegészítések pedig egyszerűen egy mellékletként a modellhez csatolhatók.

A következőkben összefoglaljuk a *Q* módszer attribútumokkal kapcsolatos legfontosabb értelmezéseit.

csak olvasható attribútum

A nem saját láthatósági szinten megadott attribútumok a lekérdező és beállító műveleteken keresztül érhetők el és alapértelmezésben csak olvashatók. Maga az attribútum ekkor saját *(private)*, esetleg csomag láthatóságú, a jelölt láthatóság pedig a műveletekre vonatkozik. A „csak olvasható" jelleget kihangsúlyozhatjuk a `{readOnly}` (vagy `{ro}`) megszorítással.

```
+a: X
```



```
-a: X
+getA(): X                  -- lemásolja az értéket
+theA():Reference<X>        -- módosítható érték
+getTheA(): X               -- gyors elérés
```

A `getA()` az értékről egy másolatot készít, amely más helyen anélkül módosítható, hogy az eredeti értéket megváltoztatnánk. A `theA()` olyan módon adja meg az értéket, hogy az a hívó által a későbbiekben is módosítható. Ez lehet például egy művelet be-/kimeneti `(inout)` paramétere. A `Reference<X>` (*„reference of X"*, `X`-re hivatkozás) valami olyan, például objektumként megvalósítható dolgot jelöl, melyen keresztül az `X` értéke elérhető, lekérhető `(get)` és beállítható `(set)`.

A rendszerről feltételezzük, hogy a csak kimeneti `(out)` paramétereket megfelelően tudja kezelni, úgy, hogy a művelet végrehajtását követően az értékek megjelennek a paraméterekben.

A `getTheA()` az érték „gyors" elérését teszi lehetővé, amit azonban csak „konstans" helyen szabad felhasználni, ahol nem módosítunk az értéken. Ilyen például egy művelet csak bemenő `(in)` paramétere. A `getTheA()` elemi érték esetén megfelel a `getA()`, összetett érték esetén pedig a `theA()` műveletnek, így annak mindig automatikusan generálható a definíciója. Megfelelő OO nyelv az érték elemi vagy összetett jellege alapján ki tudja választani a gyors elérés módját, így felesleges a `getTheA()` művelet.

Jegyezzük meg, hogy az elemi értékeknek a későbbiekben módosítható elérését nem minden környezet teszi közvetlenül lehetővé. A C++-ban rendelkezésre áll ez a lehetőség, a Java nyelvben azonban nem.

írható és olvasható attribútum

Az írható és olvasható attribútumot a definíció után írt `{read-write}` (vagy `{rw}`) megszorítással, vagy röviden a ◁ (szövegesen: „<") szimbólummal jelölhetjük, például: `+a: X◁`.

Az írás lehetősége egy újabb műveletet is jelent:

```
+setA(a_: X)
```

Az `a_` olvasása: „`a` vessző". Helyette a `newA` paraméternév is használható, az *„a vessző"* azonban lekérdezések, például egy összehasonlítás esetén is alkalmas az attribútum és a rá vonatkozó paraméter kapcsolatának jelzésére.

csak írható attribútum

Ha nem engedélyezzük az attribútum elérését és olvasását, csak a beállítását, akkor a `{writeOnly}` (vagy `{wo}`) megszorítást, vagy a rövidebb ◁◁ (szövegesen: „<<") szimbólumot használjuk.

```
+a: X◁◁
```

↯

```
-a: X
+setA(a_: X)
```

A szimbólumok vagy a megszorítások paraméterek esetén is alkalmazhatók az `in`, `out` vagy `inout` jelleg helyett. Figyeljük meg, hogy paraméterek esetén is az alapértelmezett a „csak olvasható" jelleg.

logikai értékű attribútum

Logikai értékű attribútumok esetén a lekérdező műveleteknél a `get` helyett `is` prefixet alkalmazunk, valamint felveszünk egy paraméternélküli beállító és egy törlő műveletet is, ha az attribútum írható.

```
+empty: Bool◁
```

↯

```
-empty: Bool
+isEmpty(): Bool
+theEmpty(): Reference<Bool>
+getTheEmpty(): Bool
+setEmpty(empty_: Bool)
+setEmpty()     {setEmpty(true);}
+clearEmpty()   {setEmpty(false);}
```

Egy logikai attribútumnak gyakran csak az igaz értékre történő beállítását engedélyezzük, de a törlését, illetve a tetszőleges értékre történő változtatását megtiltjuk. Ezt az esetet a `{setOnly}` megszorítással, vagy röviden a + szimbólummal jelezhetjük.

```
+empty: Bool+
```

↯

```
-empty: Bool
+isEmpty(): Bool
+setEmpty()
```

A csak törölhető attribútum a `{clearOnly}` megszorítással, vagy röviden a – szimbólummal hangsúlyozható. A csak beállítható vagy csak törölhető attribútum esetén az érték lekérdezését a szimbólumok elé írt ◁ jelölésével (például ◁+) tilthatjuk meg.

származtatott attribútum

A *Q* módszer eltér a származtatás „/" jelének hagyományos értelmezésétől is, amelyet bizonyos „közvetettségként" értelmez. Származtatott attribútum esetén ez azt jelenti, hogy ténylegesen attribútumot nem veszünk fel, csak lekérdező és esetleg beállító műveleteket.

```
+/systemTime: Time◁
```

↳

```
+getSystemTime(): Time
+getTheSystemTime():Time
+setSystemTime(systemTime_: Time)
```

Származtatás esetén a `theA()` művelet csak akkor létezik, ha a származtatás nem állít elő új értéket, hanem például egy más érték elérését rövidíti le. Ekkor a `getTheA` megegyezik a `theA` művelettel, egyébként pedig a `getA` művelettel. Az elérés-jellegű származtatást a kompozíció (lásd később) jelölésével adhatjuk meg.

Származtatott attribútum esetén a kezdőértékadás is a megfelelő beállító művelet segítségével történik.

tárolt származtatott attribútum

Tárolt származtatott attribútumok jelzésére a *Q* módszer a `{cached}` megszorítást ajánlja és a következő értelmezést (a `compute` és `invalid` prefixű műveleteket a származtatott attribútumra kiható változtatások közvetlenül hívhatják):

```
+/size: Size {cached}
```

↳

```
-size: Size
-validSize: Bool
+computeSize(): Size
+invalidSize() { validSize=false; }
+getSize(): Size
```

A `getSize` művelet Java és C++ nyelven a következő módon valósítható meg:

```
if(!validSize) { size=computeSize(); validSize=true; }
return size;
```

A *Q* módszer azt is megengedi, hogy az objektumok határain kívül az attribútumra annak alapnevével hivatkozzunk, mely ténylegesen a megfelelő beállító vagy lekérdező műveletet helyettesíti. Például az `x.a=y.b` a következőképpen értelmezhető: `y.setA(y.getTheB())`.

lekérdezés

Műveletek esetén a származtatás „/" jelét a lekérdezések jelölésére is felhasználhatjuk. A jelölést úgy értelmezzük, hogy ekkor az objektumból származtatunk egy értéket.

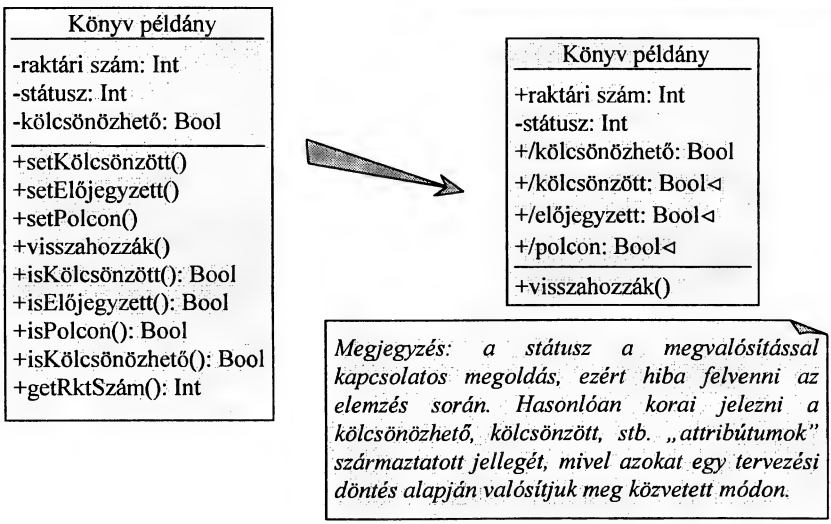
alapértelmezések

Mielőtt megrémülnénk a megvalósítandó nagymennyiségű művelettől, vegyük észre, hogy minden művelet alapértelmezett változata a jellemzők figyelembe vételével automatikusan generálható. Származtatott attribútumnak a `getA()`, illetve tárolt esetben a `computeA()` műveletét a megfelelő megszorítás határozza meg. Csak akkor szükséges tényleges kódot írnunk, ha az alapértelmezett művelettől el kívánunk térni.

Alkalmas OO nyelv esetén egyes osztályoknál, műveleteknél, illetve attribútumoknál bejelölhetjük, hogy azok optimalizálhatók. Ilyen például a `C++` nyelvnek a műveleteknél megadható `inline` jelölése, illetve bizonyos esetekben a Java `final` kulcsszava. A kifejtés technikailag úgy történik, hogy a nyelv a művelet metódusát behelyettesíti a hívás helyére. Ekkor nem történik tényleges alprogramhívás, ezért a hagyományos programozással azonos hatékonyságú kódot kapunk, amit azonban egy áttekinthetőbb és kezelhetőbb módon adtunk meg. Az optimalizálás szükségességét és lehetőségét azonban csak a tervezés utolsó szakaszában szabad megvizsgálni.

Első pillantásra kissé ezoterikusnak tűnhetnek a jelölések és az értelmezések. Segítségükkel (és a ***Q*** módszer többi megoldásával) azonban a modelljeinket áttekinthetőbbé és eredeti méretük töredékére tudjuk egyszerűsíteni.

64. Ábra: A konvenciók alkalmazása



A gyakorlati tapasztalatok alapján az egyszerűsödés szinte az összes modellre igaz. Ez pedig arra utal, hogy ez a szemlélet egy magasabb absztrakciós szintet képvisel, ugyanakkor a világról kialakított képünkhöz is közelebb áll.

Jellegzetességek

Az attribútumokat és a műveleteket közös néven jellegzetességeknek nevezzük. A modellben felvett osztályok különböző részletezettségi szinten is ábrázolhatók. A jellegzetességekre közös sztereotípia és megszorítás is megadható, valamint az attribútumok és műveletek mellett további részek is felvehetők. Osztály és attribútum esetén jelölhető a változás lehetősége.

jellegzetesség és felelősség

Az osztály attribútuma és művelete egyaránt az osztály *jellegzetessége* `(feature)`, melyeket a névrész alatt sorolhatunk fel. A jellegzetesség mellett elterjedt még a *felelősség* `(responsibility)` elnevezés is, melyet abban az értelemben használunk, hogy az alkalmazás megfelelő működéséhez az osztály felelőssége, hogy rendelkezzen az egyes jellegzetességekkel. „Felelősségként" ezért általában az osztály magasabb rendű céljára utalunk, melyet esetenként egy, de általában több attribútummal és művelettel valósítunk meg.

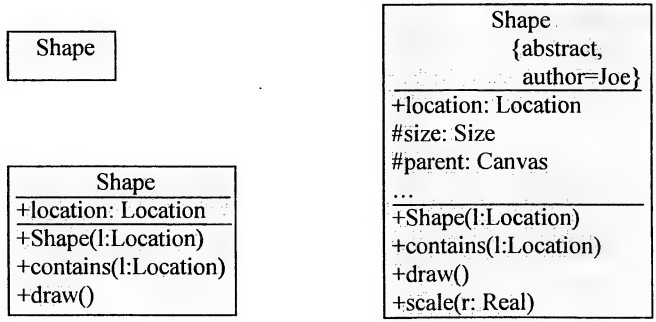
Egy attribútum az osztály egyszerű adatszolgáltatás vagy adattárolás jellegű felelősségét jelenti, míg a művelet az osztály céljának megvalósításához szükséges utasítássorozat.

Részletezettség szintjei

vázlatosabb és részletesebb diagramok

A modellünk elemeiről szemléltetésként több különböző, vázlatosabb vagy részletesebb diagramot is készíthetünk. Az osztályok a diagramokon különböző részletességgel ábrázolhatók. Így egyes attribútumok és műveletek elrejthetők, vagy csak bizonyos elemek (például az elemzés során azonosítottak) jeleníthetők meg.

65. Ábra: Részletezettség szintje



„..."

Az attribútumok és műveletek utolsó elemeként megadott három pont („...") azt jelöli, hogy a modellünkben további elemek is léteznek, de azokat a diagramon nem jelenítjük meg. A jelölés tehát nem arra utal, hogy a modellbe a fejlesztés későbbi szakaszában várhatóan még további elemeket is felveszünk.

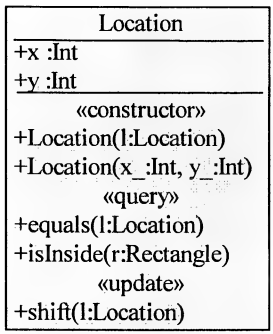
Közös sztereotípia és megszorítás

Az attribútumok és műveletek esetén sztereotípia (a definíció előtt) és megszorítás (a definíció végén) is megadható.

A jellegzetességek között egy sztereotípiát önálló elemként is felvehetünk, mely ekkor az összes ezt követő elemre vonatkozik, amíg nem szerepel egy újabb, önálló sztereotípia. Hasonló elv megszorításokon esetén is alkalmazható.

önálló sztereotípia és megszorítás

66. Ábra: Közös sztereotípia



Részek

Az attribútumokat és műveleteket általában az osztály névrésze alatt, egy-egy vízszintes vonalat követően soroljuk fel. Az UML megengedi, hogy a modellelemek névrésze alatt önállóan is definiáljunk *részeket* (`compartment`). Az egyes részek el is nevezhetők, mely elnevezést kiemeljük (általában félkövér betűtípussal) és középre igazítjuk. Osztály esetén az `attributes` *(attribútumok)* és `operations` *(műveletek)* elnevezésű részek előre definiáltak. Használati eseteknél az UML az `extension points` (kiterjesztési pont) részt definiálja.

részek

attributes operations extension points

Az UML gyakorlatában osztályok esetén a következő kiegészítő részeket alkalmazzák:

- `responsibilities` *(felelősségek)*: ebben a részben az osztály célját egy magasabb absztrakciós szinten fogalmazhatjuk meg. Az egyes felelősségeket külön sorokba írjuk.

  responsibilities

- `exceptions` *(kivételek)*: olyan üzenetek, amelyekkel az osztály a normál működésétől eltérő hibáról vagy különleges helyzetről értesíti a hívót. A kivételek adatjellegű üzenetek, melyeket általában osztályként definiálunk. Osztálydiagramokon a kivételként használt osztályokat «`exception`» sztereotípiával jelöljük.

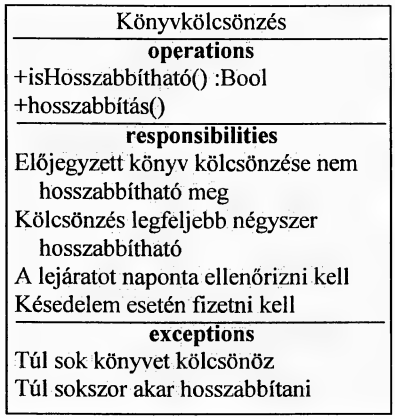
  exceptions

- `signals` *(jelzések)*: olyan üzenetek, amelyeket az osztály a normál működése közben is tud fogadni. Jelzés vételekor az osztály félbeszakítja normál működését, végrehajtja a jelzésnek megfelelő tevékenységet, majd visszatér a normál működéséhez. A modellben a

  signals

fogadható jelzéseket a műveletekkel azonos szintaktikával adunk meg, melyet a «signal» sztereotípiával jelölünk.

67. Ábra: Részek és elnevezésük



Változatlanság

Osztály vagy attribútum módosításának lehetőségét az UML-ben megszorítással jelölhetjük:

- A `{changeable}` („változtatható") megszorítás esetén az osztály objektuma vagy az attribútum értéke változtatható. Ez az alapértelmezett, ezért ezt csak akkor használjuk, ha a módosítás lehetőségét hangsúlyozni szeretnénk.

{frozen}

- A `{frozen}` („befagyasztott") megszorítással jelölhetjük, hogy az elem a kezdőbeállítása, inicializálása után nem változtatható.

A `{frozen}` értelmezése megegyezik a legtöbb nyelv konstans fogalmával, így a C++ `const`, és bizonyos helyzetekben a Java `final` jelölésével.

A változatlanság nem azonos a „csak olvasható" *(`read only`)* tulajdonsággal. A csak olvasható elem nem változtatható meg, de módosulhat más értékek változásának hatására. Ilyenek például egyes származtatott attribútumok.

Jellegzetességek elérése

jelölés

A jellegzetességek elérésére az UML kettős jelölést használ. Ha egy osztály attribútumára vagy műveletére hivatkozunk, akkor azt a csomagoknál már megismert `::` („négypont") segítségével tehetjük meg, például: `Mailer::send()`.

Ha egy osztály objektumának az attribútumértékét szeretnénk elérni, vagy az objektumra alkalmazni akarjuk egy műveletét, akkor az objektum elnevezését és a jellegzetességet ponttal kapcsoljuk össze, például `t.időpont`, vagy `t.véglegesítés()`.

A két változat még a C++ jelöléseit követi, a Java nyelvben azonban mindkét esetben használható a pont.

A ***Q*** módszer az osztály jellegzetességére történő hivatkozás esetén is engedélyezi a pont használatát.

Jellegzetességek keresése

*A **Q** módszer ajánlása szerint az osztályok alapvető jellegzetességeit a felhasználói felületek segítségével kereshetjük meg. Az attribútumok adatszolgáltató vagy adattároló felelősségként, az osztályra vonatkozó vezérlő elemek pedig műveletekként jelennek meg. A további belső jellegzetességeket a felelősségek megosztásával határozhatjuk meg.*

A „honnan jönnek az osztályok" mellett az OO módszerek másik nagy kérdése a „honnan jönnek az attribútumok és a műveletek". A hagyományos strukturált megközelítéssel ellentétben itt az egyik legnagyobb kihívás, hogy az alkalmazás együttes működését szét tudjuk osztani az objektumok között. Megfelelő tervezés esetén a működés teljes „terhe" kiegyensúlyozottan oszlik meg az együttműködő objektumok között.

Adatszolgáltató és tároló felelősségek

osztályok keresése

A **Q** modellező módszer ajánlását követve a szakterületi modellünk kiindulási osztályait a felhasználói felületek vizsgálata alapján gyűjtsük össze. A felületeken keressük meg, „rajzoljuk körbe" az összetartozó adatcsoportokat, az „információs csomópontokat", majd az ott megjelenő entitásnak a szakterület szótárából keressünk ki egy megfelelő elnevezést. Az összetartozást, az adatok közötti kohéziós erőt megfelelő számú és kellően változatos példák esetén az invariancia-elv segítségével határozhatjuk meg.

attribútumok keresése

Az osztályok felvételével párhuzamosan a felhasználói felületekről kigyűjthetjük az entitásokat jellemző attribútumokat is. Az osztállyal kapcsolatban az alkalmazás által szolgáltatott vagy kért információkat nevezzük el a megfelelő szakterületi fogalmak alapján és jelöljük attribútumokként. Ebben a lépésben célszerű kihagyni a felületek vezérléssel kapcsolatos elemeit (Pl.: `Ok`, `Véglegesítés`), melyek vagy a navigációt teszik lehetővé vagy a használati esetek tevékenységsorozataival kapcsolatosak.

Fontos, hogy jelöljük az adatáramlás irányát, hogy az alkalmazás szolgáltatja az adatot („lekérdezhető attribútum"), kéri annak az értékét („írható attribútum") vagy mindkét eset előfordul.

Az attribútum adatszolgáltató és/vagy értéktároló felelősségként jelenik meg.

adatszolgáltatási felelősség

- Az *adatszolgáltatási* („lekérdezési") *felelősség* azt jelenti, hogy egy konkrét objektumnak meg kell tudni mondania, hogy igaz-e rá egy adott tulajdonság, vagy hogy egy adott szempontból milyen tulajdonságú (a szempont alapján milyen értékkel, azaz adattal írható le ez a tulajdonsága).

- Egy *értéktárolási felelősség* azt jelenti, hogy az objektumnak megadhatjuk, hogy arra egy adott tulajdonság teljesül, illetve be tudjuk állítani az adott szempont szerinti tulajdonság-értékét.

értéktárolási felelősség

Az adatok közötti, matematikai egyenletekkel megadható kapcsolatokat megszorításokként vehetjük fel. Az összefüggésekből kiválaszthatunk és megjelölhetünk származtatott elemeket, de ezt a döntést a tervezés fázisára is halaszthatjuk.

Az attribútumok elnevezése mellett célszerű meghatározni a felvehető adatok körét, azaz a típust. A típus lehet elemi (tovább nem bontható) vagy összetett.

Az UML a diagramokat a forráskód tervrajzának tekinti ezért a gyakorlatban általában a kiválasztott platformnak megfelelő típusokat használnak. A ***Q*** módszer az elemzés során gépfüggetlen, „koncepcionális" típusokat javasol. Ezek lehetnek például a következők:

koncepcionális típusok

- `Num:` számérték
- `Nat:` természetes szám (0 vagy pozitív egész)
- `Int:` egész szám
- `Real:` valós szám
- `Bool:` logikai érték
- `Char:` egyetlen karakter
- `Code:` kód-jellegű szöveg, például egy számsor
- `String:` rövid (egysoros) szöveg
- `Text:` hosszabb (többsoros) szöveg.
- `Date:` dátum
- `enum:` felsorolásos érték

Az UML ajánlása alapján a felsorolásos érték elemeit megszorításként adhatjuk meg (`nem: enum {férfi=1; nő}`). Hasonló módszert javasol a ***Q*** módszer a felvehető értékek és az értéktartományok pontosítására, pl.:

érték-tartományok

```
signum: Int {-1,0,+1}
hétköznap: Nap {hétfő..péntek}
tél: Hónap {december,január..február}
x: Real {-1..+1}
```

A koncepcionális típusokat a fejlesztés későbbi fázisában, a tervezés során képezzük le a kiválasztott platform típusaira.

Entitások és értékek

Az objektumok egyik alapvető tulajdonsága az egyediségük. Ebben különböznek az értékektől, amelyek az entitásra vonatkozó, de attól már elszakított adatot határoznak meg. Az értékek szabadon másolhatók, míg egy entitás többszörözése könnyen a rendszer inkonzisztenciájához vezethet.

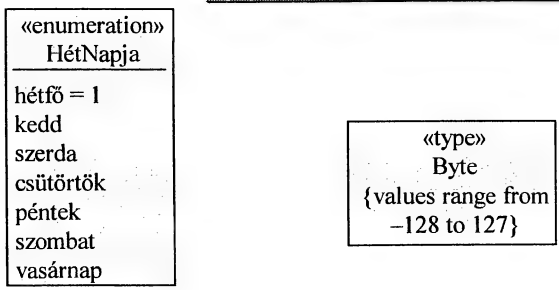
entitás- és érték-objektum

Ha eltekintünk az egyediségtől, akkor az osztályok az összetett értéktípusok ábrázolására is alkalmasak. Ha szükséges, akkor megkülönböztetésül az *entitás-objektum* és az *érték-objektum* (`value-object`) elnevezéseket használjuk.

Az érték-objektumokat az elemi értékekhez hasonlónak tekintjük, melyek nem rendelkeznek egyediséggel, így az inkonzisztencia veszélye nélkül másolhatók.

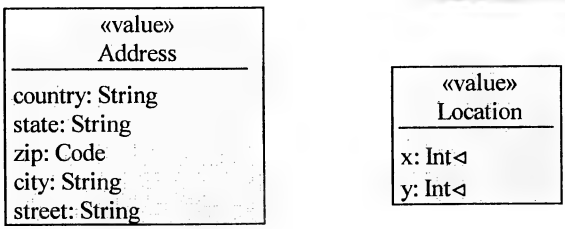
Az UML ajánlása szerint az elemi és az összetett értékek típusát «`type`» sztereotípiával jelölt osztályokként reprezentálhatjuk. Az elemi típus értéktartományát megszorításként, szövegesen adhatjuk meg. A felsorolásos típusok sztereotípiája az «`enumeration`», a lehetséges értékek elnevezéseit pedig attribútumként vehetjük fel. A kezdőértékadás szintaktikájának megfelelően az egyes elnevezésekhez tartozó kódolást is definiálhatjuk.

68. Ábra: Felsorolásos és elemi típus (UML)



Az UML a «`type`»-ot más értelemben is használja, ezért a *Q* módszer az érték-jellegű típusok jelzésére az «`enumeration`» mellett a «`value`» sztereotípiát ajánlja.

69. Ábra: Érték-típus (Q)



Az alapvető szakterületi osztályok és azok attribútumainak összegyűjtése után határozzuk meg a felvehető értékek típusát. Keressünk összetartozó, de egyediséggel nem rendelkező értékeket, melyeket összetett értéktípusokba csoportosítunk, és az osztálydiagramon értékjellegű osztályokként ábrázolunk.

Műveletek keresése

használati esetek és felhasználói felületek alapján

A felhasználói felületek vizsgálatával már meg tudunk határozni néhány osztályt, illetve azok adatszolgáltató és értéktároló felelősségeit. Az alkalmazás működéséhez szükséges további felelősségeket úgy kereshetjük meg, hogy sorra vesszük a használati eseteket, valamint a felhasználói felületeknek az osztályokra vonatkozó minden elemét, és megvizsgáljuk, hogy azokat mely osztály milyen már létező vagy újabb felelősségével (műveletével vagy egyszerűbb esetben attribútumával) lehet megvalósítani.

Egy osztály újabb felelősségének felvételekor nagy vonalakban elképzeljük a megvalósítás módját. A megvalósítás során megkíséreljük a felelősséget részben vagy egészben más osztályra áthárítani, tehát soha nem avatkozunk be más osztály „belügyeibe".

delegáció

Ha egy művelet működése alapvetően arra korlátozódik, hogy önmagának vagy egy más osztálynak a hasonló jellegű műveleteit meghívja, akkor azt a művelet továbbításának, *delegációjának* `(delegation)` nevezzük. A delegáció is a felelősség áthárítása, megosztása.

A felelősségek keresésének ez a módszere az objektumorientáltság történetének egy régi, egyszerű, de nagyon hatékony technikáján, a CRC kártyák módszerén alapszik.

CRC-kártyák

osztály felelőssége

A *CRC-* `(Class-Responsibility-Collaboration,` *osztály-felelősség-együttműködés)* kártyák módszerét a `Smalltalk` programozás, illetve az objektumorientált tervezés tanítása során fejlesztették ki. A technikát Cunningham és Beck publikálták a '80-as évek végén. Az osztály felelősségének az osztály céljának magas szintű leírását nevezzük, melyeket a későbbiekben attribútumokként vagy műveletekként valósíthatunk meg.

70. Ábra: CRC-kártya

Jelentkezés	
kapcsolattartás adatai	
jelentkezők száma	
ár	Tanfolyam
értesítés a tanfolyamról	Tanfolyam
jelentkezés lemondása	

A módszer alkalmazásakor minden osztályt egy-egy kártyával reprezentálunk, melyen csak annyi részletet szabad megadni, amennyi még ráfér a kártyára.

A kártya tetejére felírjuk az osztály nevét, majd minden egyes sorban megadunk

- egy felelősséget egy kifejezéssel, valamint
- a felelősség megvalósításához szükséges együttműködő osztály (ok) nevét is felsoroljuk.

egyszerű módszer

A CRC-kártyák módszerének legnagyobb előnye, hogy segítségével szinte a végletekig egyszerű, mégis „formalizált" módon gyűjthetjük össze az osztályok felelősségeit. Felhasználásával egyszerűsíthetjük a fejlesztők közötti megbeszéléseket (`„brain storm"`) és konkretizálhatjuk a feladatokat.

legfeljebb négy-öt felelősség

Az áttekinthetőség miatt a kártyákra csak magasszintű felelősségeket szabad felvenni és kerülni kell a részletezéseket. Az esetek többségében az osztály céljait egy-két, de legfeljebb négy-öt felelősségben célszerű összefoglalni. Ha több felelősség gyűlt össze, akkor kíséreljük meg a szorosabban összetartozó elemeket egy összefoglaló kifejezéssel helyettesíteni.

Felelősségek és belső osztályok keresése

felhasználói felületek és használati esetek vizsgálata

A ***Q*** modellező módszer a CRC-kártyákhoz hasonló technikával gyűjti össze az osztályok további felelősségeit, attribútumait és műveleteit. A felhasználói felületek vizsgálatával megkereshetjük az alapvető osztályokat és azok attribútum-jelegű jellemzőit. A következő lépésben a felhasználói felületek és a használati esetek segítségével összegyűjtjük a vezérléshez szükséges műveleteket.

Az egységesség érdekében a ***Q*** módszer minden csomagot egy-egy osztálynak tekint, mely így saját attribútumokkal és műveletekkel is rendelkezhet. Egy csomagot és a benne elhelyezkedő, azonos nevű, de nagybetűvel kezdődő osztályt ugyanannak a modellelemnek tekintjük.

A felhasználói felületek az alkalmazás külső kapcsolódási protokollja, melyek az alkalmazás belső világán belül tárolt szakterületi modell külső, kivetített képei. A felhasználói felületeken található elemek legnagyobb része a következő csoportokba sorolhatók:

- Az információk értelmezését teszik lehetővé bizonyos *feliratok* és kiegészítő, „díszítő" szövegek, melyek az adattömegben való tájékozódást segítik. A fejlesztés szempontjából ezek csak megjegyzések.
- A *navigációs elemek* a felhasználói felületek közötti mozgást teszik lehetővé. A felhasználói felületek összefoglaló ábráján az egyes elemeket ezek a vezérlések kapcsolják össze.
- A *változó elem* a megfelelő (a körberajzolással már meghatározott) objektum attribútum-jellegű jellemzőjének a külső képe, mely lehet olvasható és/vagy beállítható.
  - A változó elem közölheti/beállíthatja a megfelelő attribútum értékét. Az érték lehet logikai is.
  - A változó elem közölheti, hogy a megfelelő attribútum egy adott értékű-e (egyenlőségvizsgálat), vagy az attribútumot beállíthatja egy adott értékre. Ez az elem is logikaiként jelenik meg, de ez több „származtatott" logikai attribútum rövidítése.
- A *vezérlő elemek* segítségével a megfelelő objektumot vezérelhetjük, annak elindíthatjuk egy módosító (nem lekérdező) műveletét.

A navigáció kiválthat egy lekérdezést, beállítást vagy vezérlést, illetve a változó és vezérlő elemek is kiválthatnak navigációt.

„horgony"

Adottnak tekintjük, hogy a felhasználói felületek minden egyes eleme eléri a hozzá tartozó „körberajzolt" objektumot. Képszerűen: minden elem egy-egy horgonyt akaszt a megfelelő objektumba. Ha a horgony valamilyen okból kifolyólag éppen nincs meghatározva, „szabadon lóg", akkor a felület eleme nem aktív.

Először tehát nincs más dolgunk, mint hogy a felhasználói felületek minden eleme esetén megkeressük a horgonynak megfelelő objektumot. Az objektum lehet egy entitás, de reprezentálhatja magát az alkalmazást, vagy a csomagot is. Ha még nem létezik ez az objektum, akkor határozzuk meg és vegyük fel az osztályát. Ezután vizsgáljuk meg, hogy a felület kiválasztott elemének megfeleltethető jellegzetességgel rendelkezik-e már az osztály, és ha igen, akkor az alkalmas-e az elem által megkövetelt célra. Ha még nem létezett a szükséges attribútum vagy művelet, akkor azt vegyük fel a megfelelő osztályban.

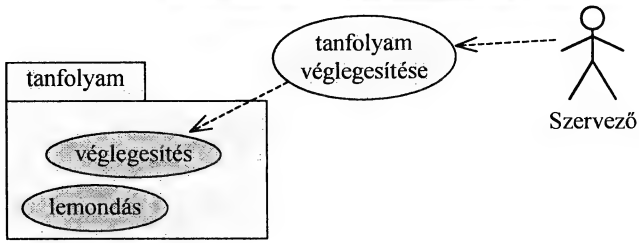
Az elemzés kezdetén az áttekinthetőség érdekében az objektumok struktúrájával kapcsolatos műveleteket (az objektumok elérése, felvitele és törlése) adottnak tételezzük fel.

használati esetek és jellegzetességek

Megfigyelhetjük, hogy a csomagok néhány használati esete a csomagot reprezentáló osztály, vagy a megfelelő entitásosztály jellegzetességeként jelenik meg, például a `tanfolyam meghirdetése` használati esetnek a `Tanfolyam` osztály `meghirdetés` művelete feleltethető meg. A használati esetek és a jellegzetességek közötti kapcsolat nem véletlen, mivel a használati eset egy olyan funkcionális követelmény, mely a többi követelményhez, felelősséghez hasonlóan adott jellegzetességekkel valósítható meg. A ***Q*** módszer értelmezésében a használati eset egy olyan művelet, amely lehet „koncepcionális", így nem feltételül jelenik meg közvetlenül, hanem csak több jellegzetesség adott módú felhasználásaként. Ugyanakkor az osztály műveleteit az osztály használati eseteinek tekinti.

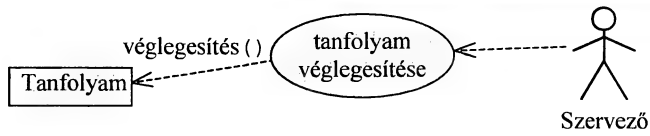
Az UML megengedi, hogy egy osztályra is megadjunk önálló diagramként használati eset diagramot. A ***Q*** módszer kiterjesztésében az egyes osztályokat az osztállyal megegyező nevű, de kisbetűvel kezdődő csomagként is megjeleníthetjük, melyben aldiagramként megadhatjuk a használati eseteit. A használati eseteket a már ismert módon összekapcsolhatjuk.

71. Ábra: Osztály műveletei, mint használati esetek



Másik jelölési módként a „csomag" alakzatában végződő szaggatott vonalú nyíl mellé írhatjuk a megvalósításhoz szükséges használati eset, azaz a művelet nevét, melyet zárójelpárral zárunk. A jelöléssel áttekinthető módon ábrázolhatjuk a műveletek egyszerű továbbítását, a delegációt is.

72. Ábra: Osztály felhasznált művelete



felelősségek megosztása

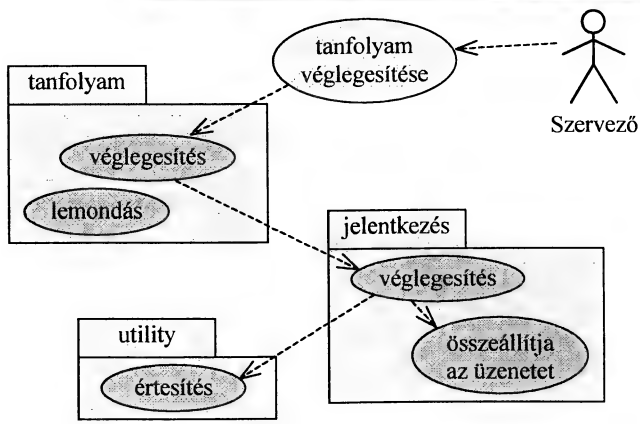
Minden újabb felelősség felvételekor vagy módosításakor gondoljuk végig, hogy azt milyen lépések segítségével tudjuk végrehajtani. A legtöbb lépés vázlata megtalálható a használati esetnek megfelelő forgatókönyvekben, illetve működési leírásokban. Több objektumra is vonatkozó összetett műveletek esetén soha ne avatkozzunk be más objektumok „magánügyeibe", hanem kíséreljük meg a felelősséget megosztani, áthárítani a további fogalmakat reprezentáló osztályokra.

- Ha az osztálynak már volt ilyen felelőssége, akkor ellenőrizzük, hogy az alkalmas-e a kívánt működésre.
- Ha az együttműködőben még nem volt ilyen felelősség, akkor azt felvesszük és végiggondoljuk, hogy az önállóan végrehajtható-e vagy felhasználható egy másik osztály a felelősség megvalósítására.

Bonyolultabb, összetettebb lépéssorozattal megvalósítható műveletek esetén ismételten megpróbáljuk a felelősséget megosztani, áthárítani más osztályokra vagy ugyanazon osztály más felelősségére.

Például egy tanfolyamot úgy véglegesíthetünk, ha sorra veszünk minden egyes érvényes (nem lemondott) `Jelentkezést`, és azt is megkérjük a véglegesítésre. A jelentkezés a véglegesítés során összeállít egy e-mail üzenetet, melyet a `utility` csomag `értesítés` műveletével küld el.

73. Ábra: Felelősségek megosztásának ábrázolása (Q)



újabb felelősségek és osztályok

A felelősségek megosztása során így megjelenhetnek újabb felelősségek. Bizonyos felelősségek azonban új osztályokat is igényelhetnek. Ebben az esetben keressük meg a megfelelő szakterületi fogalmat és adjuk meg osztályként.

Fontos, hogy a szakterületi elemzés során kizárólag a szakterületi fogalmaknak megfelelő osztályokat és felelősségeket vegyük fel, azaz a modellünket csak addig a szintig pontosítsuk, amíg az a felhasználó számára is ismert és így meg is érthető.

A művelet alanya és tárgya

Egy osztály művelete egyaránt jelölhet olyan tevékenységet, amelyet az osztály objektuma hajt végre, vagy egy olyan transzformációt, amelyet az objektum szenved el. Az egyértelműség kedvéért néha célszerű jelölni, hogy az objektum a művelet alanya vagy annak tárgya. Egy `Kutya` osztály `harap` művelete esetén nem mindegy, hogy a kutya harap-e vagy a kutyát harapják meg (főleg a kutya számára).

Az osztály módosító (nem lekérdező) műveleteit a ***Q*** modellező módszer további két altípusra bontja.

- Az *aktív művelet* az objektum összetett viselkedésének egy része, amelyet az objektum hajt végre, így maga az objektum a művelet alanya. Az osztály aktív műveleteit igeként nevezzük el, például `harap`, `értesít`, `véglegesít`, stb. módon.

aktív művelet

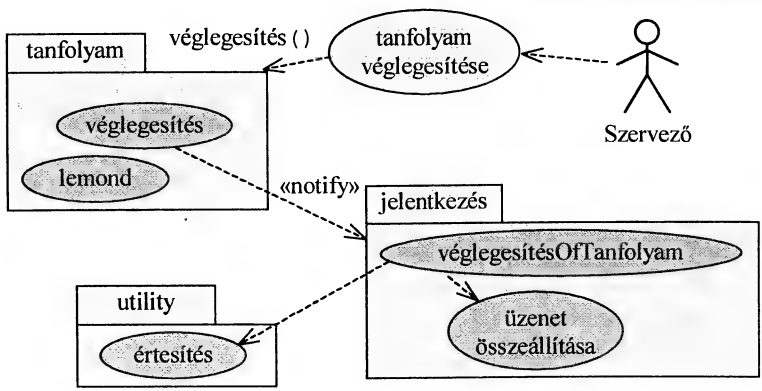
- A *passzív művelet* egy külső hatás felvételét jelenti, mely hatásra az osztály objektuma megváltozhat. Az osztály objektuma a passzív művelet tárgya. Az osztály passzív műveleteit az igéből képzett -ás, -és végződésű főnevekkel nevezzük el, például `harapás`, `értesítés`.

passzív művelet

Bizonyos műveletek más objektum változását is eredményezhetik. A közvetett hatásokat az értesítés *(notify)* alapú szerkezetekkel jelölhetjük. Ha egy név nélküli «notify» sztereotípiájú szaggatott vonalú nyíl egy `Bb` osztály `aa` műveletéből indul ki és egy osztályban végződik, akkor úgy tekintjük, hogy az `aa` vagy `aaOfBb` műveletet hívja meg, ahol `aa` az eredeti művelet, `Bb` pedig a kiindulási osztály neve. A láncban lezajló értesítések során általában nem halmozzuk a prefixeket, hanem vagy a módosító műveletet, vagy az eredeti művelet nevét használjuk fel.

«notify»

74. Ábra: Passzív műveletek (Q)



Kommunikációs útvonalak

Az objektumnak el kell érni a vele együttműködő objektumokat. Az objektumok közötti kommunikációs útvonalakat, információs csatornákat kapcsolatoknak nevezzük. Az osztály a kapcsolatot strukturális módon is biztosíthatja, amelyet asszociációnak nevezünk. Az asszociáció végei a szerepek, melyeknek jelölhető a számossága, hogy egy kapcsolatban a szerep hány objektuma vehet részt.

kapcsolat

A felelősségek megosztása során az objektumnak el kell tudni érnie az együttműködő osztályokat. A *kapcsolatok* `(link)` olyan útvonalakként tekinthetők, amelyeken keresztül az objektumok információkat továbbíthatnak, például a felelősségek is ezeken az útvonalakon terjedhetnek. Például a `Tanfolyam` osztálynak ismernie kell a `Tanfolyam típus` és a `Jelentkezés` osztályokat ahhoz, hogy a felelősségeit megvalósíthassa.

strukturális kapcsolat

elérés felelőssége

Ha az együttműködő objektum műveletének meghívásáig, azaz a delegáció pontjáig nem érhető el a felelősség megvalósításához szükséges objektum, akkor magának az osztálynak kell strukturális módon biztosítani a kapcsolatot. A strukturális kapcsolat egyben a szakterületi modell fogalmai közötti kapcsolatot is ábrázol. Strukturális kapcsolattal megvalósítható az *„elérés felelőssége"*. A kapcsolat nem attribútum, mivel nem egy érték jellegű fogalmat ír le, hanem két entitás közötti viszonyt. Ugyanakkor a kapcsolat olyan adattagként is megvalósítható, amely alkalmas a másik objektum objektumazonosítójának (OID) tárolására.

Kommunikációs út

A felelősségek megosztása, így a delegáció is igényli azt, hogy a művelet végrehajtásában együttműködő objektumok információt közölhessenek, azaz kommunikálhassanak egymással.

Alapvetően két fajta módon biztosítható az együttműködő objektum elérése, ha a művelet adott pontján az nem áll rendelkezésre:

visszafelé hivatkozás

- Lehetséges, hogy a használati esetet megvalósító korábbi műveletek ismerik, el tudják érni azt az objektumot, amelyre a hívott műveletnek szüksége van. Ekkor valamely korábbi művelettől kezdve paraméterként „leküldhetjük" az objektumot, így a hívott művelet el tudja érni az együttműködőjét. Ha a paraméterként megkapott objektum a hívási lánc egy korábbi objektuma, akkor azt *visszafelé hivatkozásnak* `(backward reference)` nevezzük.
- Ha a hívási lánc egyetlen korábbi művelete sem ismeri az igényelt objektumot, akkor egy strukturális kapcsolattal kell biztosítani az elérhetőséget.

Asszociáció és kapcsolat

kapcsolat

Konkrét objektumok között az elérés lehetőségét *kapcsolatnak* `(link)` nevezzük. Az egymással kapcsolatban álló objektumok matematikailag egy elem n-essel adhatók meg, például a `CompSoft` cég és `Nagy István` közötti `Alkalmazás` kapcsolat az `Alkalmazás(CompSoft, Nagy István)` elempárral. Az objektum eléri például a paraméterként megkapott, vagy az általa újonnan előállított másik objektumot. *Strukturális kapcsolatnak* nevezzük azt az esetet, ha az elérést maga az objektum szolgáltatja, így az rendelkezésre áll minden olyan pillanatban, amikor a kiindulási objektum is elérhető. A nem strukturális, *temporális* kapcsolat ezzel szemben csak az objektum működésének bizonyos pontjain „él".

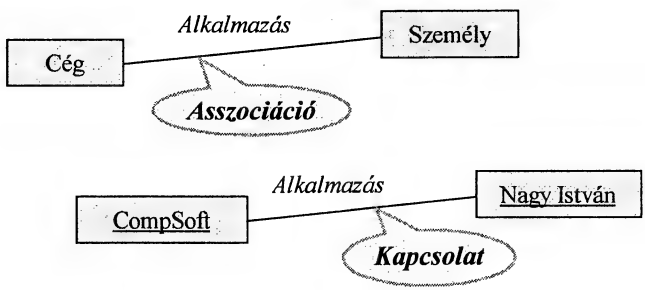
asszociáció

Az osztályok közötti *asszociációnak* `(association)` nevezzük az osztályok objektumai közötti *lehetséges* strukturális kapcsolatokat. Az asszociáció a kapcsolat mintája, ezért a kettő az osztály/objektum fogalompárhoz hasonlóan típus/példány viszonyban van egymással. Matematikailag az asszociációt az osztályok elem n-esével adhatjuk meg, például `Alkalmazás(Cég, Személy)`, azaz egy cég és egy személy `Alkalmazás` elnevezésű viszonyban állhat egymással.

bináris asszociáció

Az UML a két osztály közötti *bináris asszociációt*, a két osztályt összekötő vonallal jelöli. Az asszociációt el is nevezhetjük, mely elnevezést az UML ajánlása szerint nagybetűvel kezdjük. Osztály önmagával vett asszociációját *reflexív asszociációnak* nevezzük.

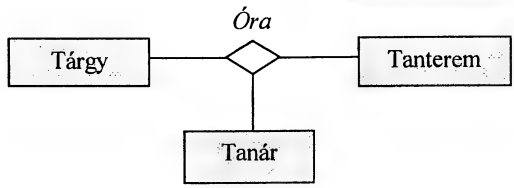
75. Ábra: Asszociáció és kapcsolat



ternáris és magasabbrendű asszociáció

A *ternáris* (három osztály közötti) vagy a *magasabbrendű asszociációt* egy rombusszal jelöljük, melyet az egyes osztályokhoz vonalakkal kapcsolunk. Ebben az esetben a nevet a rombusz mellett adjuk meg. (Az UML szerzőinek képét követve bináris esetben „0 vastagságú" vonallá nyomjuk össze a rombuszt.)

76. Ábra: Ternáris asszociáció



{implicit}

Az `{implicit}` módon megjelölt asszociáció azt jelenti, hogy a kapcsolat csak elvi, de az alkalmazásban ténylegesen nem jelenik meg.

Szerepek

szerep

számosság

Az asszociáció végeit *szerepeknek* `(role)` nevezzük, mivel a megfelelő objektumoknak a kapcsolatban betöltött szerepét jelölik. A szerep *multiplicitása* vagy *számossága* `(multiplicity)` jelzi, hogy egy konkrét viszonyban hány konkrét objektum vehet részt.

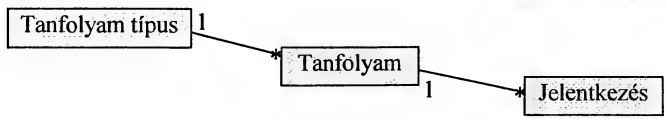
Az UML jelöléseivel az asszociáció megfelelő végéhez, a szerephez írjuk a számosságot, mely lehet egy számérték (például `3`), egy `a..b` formában megadott értéktartomány (például `1..10`), vagy ezek vesszővel elválasztott sorozata (pl.: `1,7` vagy `0,5,10..20`). A `*` a végtelent jelöli (pl.: `1..*`, azaz „egynél több"), önállóan pedig a `0..*` számosságot, azaz a *tetszőleges* számú multiplicitást adja meg.

Ternáris, vagy magasabbrendű asszociációk esetén a számosságot olyan értékként értelmezzük, amely megadja, hogy a kapcsolat többi objektumának rögzítése esetén hány kombinációban fordulhat elő a vizsgált szerepű objektum.

A leggyakrabban használt multiplicitások a következők:

`1`	(pontosan egy),
`*`	(tetszőleges, azaz nulla vagy több),
`0..1`	(opcionális, azaz nulla vagy egy)
`n`	(pontosan egy adott számérték)
`0..n`	(legfeljebb egy adott számérték)

77. Ábra: Multiplicitás



opcionális számosság

A ***Q*** modellező módszer megengedi, hogy a viszonylag gyakori opcionális számosságot a 0-val rövidítsük, melyet „o", illetve „opcionális" `(optional)` módon ejtünk ki. Az OMT a tetszőleges számosságot a vonal végére helyezett •, az opcionálist pedig a ○ (üres karika) szimbólummal jelölte. A vonal végére helyezett *, illetve 0 jellel hasonló hatást érhetünk el. Ez különösen akkor hasznos, ha nagyszámú kapcsolat esetén nehéz megállapítani, hogy melyik számosság melyik szerephez tartozik.

alap-értelmezés

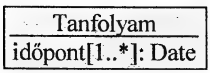
A nem megadott számosságot az UML „specifikálatlannak", meghatározatlannak tekinti, a ***Q*** módszer azonban azt 1 („pontosan egy") értékként értelmezi, mivel megköveteli, hogy a számosságokat minden asszociáció felvételekor azonnal határozzuk meg.

Attribútum és osztály számossága

A számosságot attribútumok esetén is megadhatjuk a nevet követő szögletes-zárójelek között, például `útvonal[2..*]: Hely`. Attribútumok esetén alapértelmezettnek az UML is az 1 számosságot tekinti.

attribútum számossága

78. Ábra: Attribútum számossága



A szögletes-zárójelek megegyeznek bizonyos programozási nyelvek tömb jelölésével, itt azonban mindössze arra utalnak, hogy az attribútumhoz több érték is tartozhat. Ugyanakkor az is lehetséges, hogy konkrét számértékkel megadott számosság esetén az értékek tárolását valóban tömb segítségével valósítjuk meg.

A számosság segítségével opcionális attribútumot is jelölhetünk (pl.: `name[0]: String`, kiejtve: „opcionálisan megadható név"). Az opcionális jelleget úgy értelmezzük, hogy le kell tudnunk kérdezni, hogy az attribútum meg van-e adva, fel kell készülnünk a megadatlan esetre, illetve az attribútum törlése is lehetséges. Az üres attribútum jelölése egy speciális értékkel is történhet, például szöveg esetén egy üres szöveggel (`""`). Az egy számosságú attribútumot kötelezően megadandónak *(„not null")* tekintjük.

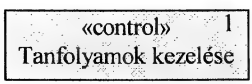
opcionális számosság

„not null"

A számosság osztályok esetén is megadható, amellyel azt jelezhetjük, hogy az alkalmazásban az osztály hány objektuma létezhet. A számosságot az UML jelöléseivel a téglalap jobb felső sarkába írjuk.

osztály számossága

79. Ábra: Osztály számossága



Ha osztály esetén nem adunk meg számosságot, akkor azt tetszőlegesnek (*) tekintjük. Gyakori még az 1 számosság, például a csomagokat reprezentáló osztályok esetén. Előfordulhat a 0 számosság is, például ha az osztálynak csak gyűjtő szerepe van, mert csak konstansokat vagy bizonyos technikákat definiál (`utility` osztály).

A ***Q*** módszer értelmezésében, ha egy önállóan szereplő osztály esetén megadunk számosságot, akkor az arra utal, hogy bizonyos helyzetekben, például a tesztelés során számolnunk kell az osztály létező objektumait és ellenőriznünk kell azok számát. A módszer más megoldást javasol, ha az osztály számossága valamely különleges ok miatt 0 vagy 1.

A ***Q*** módszer megengedi, hogy az attribútum multiplicitását a típus elé vagy mögé írjuk. Például: `útvonal: Hely[2..*]`, azaz az útvonalat legalább két hely felsorolásával adjuk meg. Osztály számosságát pedig a nevet követően, szögletes zárójelek között is felvehetjük.

Módosítás lehetősége

Az attribútumokhoz hasonlóan megszorításként az asszociációnál, illetve specifikusan a szerepnél is jelölhető a módosítás lehetősége:

- `{changeable}` *(változtatható)*: az objektumok közötti kapcsolat szabadon felvihető, törölhető és megváltoztatható. Mivel az UML-ben ez az alapértelmezés, ezért csak akkor célszerű megadni, ha hangsúlyozni szeretnénk a módosítás lehetőségét.

{frozen}

- `{frozen}` *(„befagyasztott")*: az egyszer már megadott kapcsolat nem törölhető és nem módosítható.

{addOnly}

- `{addOnly}` *(„csak bővíthető")*: a kapcsolatok legfeljebb bővíthetők, de nem törölhetők, és nem is módosíthatók.

Ha a módosítás lehetőségét a szerepnél adjuk meg, akkor az arra vonatkozik, hogy az ellentétes oldalról felvihető-e, törölhető-e, stb. objektum.

Mivel attribútumok esetén is megadhatunk számosságot, ezért ott is jelölhetjük az `{addOnly}` „csak bővíthető" jelleget.

{ordered}

Gyakori még a szerepnél megadott `{ordered}` megszorítás, amely arra utal, hogy a kapcsolatban lévő objektumokat meghatározott sorrendben érjük el.

A sorrendiségre vonatkozó `{ordered}` megszorítás jelentheti a felvitel sorrendjét is, ezért nem feltétlenül utal rendezettségre. A *rendezettség* az objektumok olyan sorrendisége, melyet végigkövetve az objektumból képezhető bizonyos érték, például valamely attribútumértéke növekszik (vagy csökken). Fowler és Scott ajánlása szerint a rendezést `{sorted}` módon jelöljük, melyet kiegészíthetünk a növekvő vagy csökkenő értékeket meghatározó kifejezéssel.

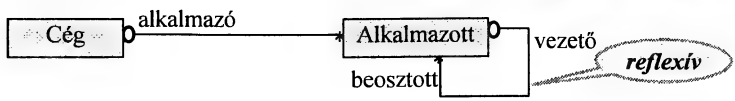
{sorted}

Elnevezések

Az asszociációk mellett a szerepek is elnevezhetők. Azonban csak akkor célszerű elnevezéseket használni, ha ezzel érthetőbbé válik a modell. Felesleges például egy asszociáció „kapcsolatban van" elnevezése. Reflexív asszociáció esetén, azaz ha az osztály önmagával van asszociációban, akkor általában fontos a *szerepnevek* `(role names)` megadása.

szerepnév

80. Ábra: Szerepnevek



szerepek láthatósága

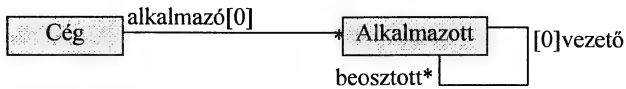
Az UML a szerepek elnevezésére kisbetűvel kezdődő, egyes számú főnevet, vagy főnévi szerkezetet ajánl. A szerephez vagy annak elnevezése elé írt +, – vagy # szimbólummal jelöljük az elérés nyilvános `(public)`, védett `(protected)` vagy saját `(private)` jellegét. A nyilvános szerep objektumai bárki által elérhetők, aki az ellentétes oldal kiindulási objektumát is eléri.

Érdemes megfigyelni, hogy a szerepeket nem az osztályból kitekintve nevezzük el, mint ahogy azt az első OO jelölések tették. A régi megoldás ugyanis nem használható ternáris vagy magasabbrendű esetekben. A szerepek elnevezését mindig az asszociáció szempontjából végezzük el. Képzeletben beállunk az asszociáció helyébe és onnan tekintjük az egyes osztályokat. A szerepnevek az objektumok kapcsolatban elfoglalt helyét írják le, például:

```
Alkalmazás(alkalmazó: Cég, alkalmazott: Személy).
```

A *Q* modellező módszer a szabványt kiegészítve megengedi, hogy a szerep számosságát a szerepnév előtt vagy mögött szögletes zárójelek között adjuk meg. A [*] tetszőleges számosság rövidíthető [] vagy * módon.

81. Ábra: Számosság (Q)



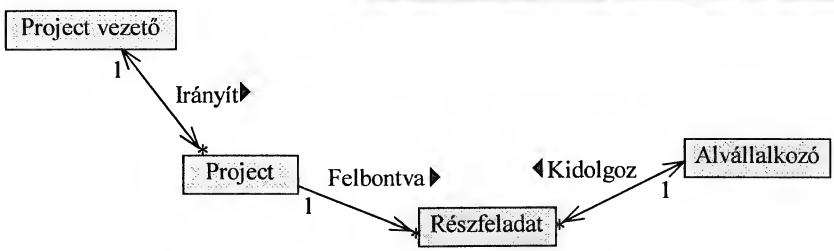
Az asszociációt elnevezhetjük egy adott szerep irányából. Az elnevezés irányát a nevet megelőző vagy azt követő fekete háromszöggel jelöljük, amely az olvasási sorrendnek megfelelő irányba mutat (például: `Alvállalkozó Kidolgoz ▶ Részfeladatot`).

elnevezés iránya

Az UML egy újdonsága, hogy az asszociációk esetén megadhatjuk a navigáció irányát *(navigability)*. A vonal végére helyezett nyílheggyel jelöljük, hogy az objektumok az ellentétes oldalakról elérhetők. Az asszociáció így lehet *egyirányú (unidirectional)* vagy *kétirányú (bidirectional)*. A jelöletlen esetet az UML kétirányúnak tekinti.

navigáció iránya, egyirányú és kétirányú asszociáció

82. Ábra: Elnevezés és a navigáció iránya



Kollekció és felsorolás

Az azonos osztályhoz tartozó objektumok egy csoportját *gyűjteménynek* vagy *kollekciónak (collection)* nevezzük. Egy kapcsolat több számosságú szerepénél elhelyezkedő objektumok ugyancsak kollekciót alkotnak. A kollekcióhoz külön, az objektumok csoportjával kapcsolatos (felvitel, törlés, keresés, stb.) műveletek és esetleg attribútumok tartoznak, ezért azt általában külön osztályként valósítjuk meg, melyet *tároló osztálynak (container class)* nevezünk.

kollekció

tároló osztály

A több számosságú szerepet az általános konvenció szerint halmaznak `(set)` tekintjük, amelyben eltekintünk az elemek sorrendjétől, illetve egy objektum abban csak egyszer szerepelhet. Az `{ordered}` megszorítás esetén lényeges a felsorolt objektumok sorrendje.

felsorolás (iterátor)

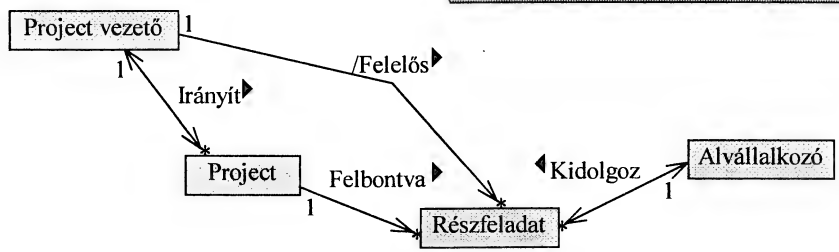
Felsorolásnak `(enumeration)` vagy *iterátornak* `(iterator)` nevezzük azt az osztályt, amely alkalmas arra, hogy egy kollekció elemeit sorra vegyük.

Származtatott és részhalmaz asszociációk

származtatott asszociáció

A *származtatott asszociációt* `(derived association)` a modell más elemeiből, például az osztálydiagram más asszociációiból számíthatjuk. A származtatott asszociációt a név elé vagy a vonalra írt / jellel jelöljük.

83. Ábra: Származtatott asszociáció



Az elérhetőség tranzitív, ezért ha egy `A` objektum elér egy `B` objektumot, az pedig egy `C` objektumot, akkor az `A` is eléri `C`-t. Csak akkor vegyünk fel ezért származtatott elemet, ha az alkalmazásunkban az valóban szükséges.

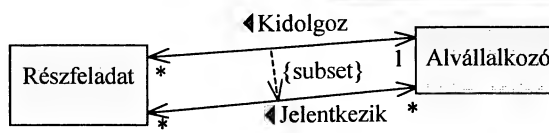
A ***Q*** módszer megengedi, hogy a származtatás jelzését a szerepnél vagy az osztálynál adjuk meg. Mindkét esetet ugyancsak származtatott asszociációként értelmez.

Gyakran előfordul, hogy egy kapcsolatban csak valamely más kapcsolatban részt vevő objektumokat engedélyezünk. Például kiköthetjük, hogy egy részfeladatot kidolgozó alvállalkozó csak a kidolgozásra jelentkezők közül kerülhet ki.

részhalmaz asszociáció

A két asszociáció közötti részhalmaz viszonyt `{subset}` („részhalmaz") megszorításként adhatjuk meg, melyet a részhalmaz asszociációból az alap asszociációig vezető szaggatott vonalú nyíl mellé írunk.

84. Ábra: Részhalmaz-asszociáció



Részhalmaz-asszociációval jelöljük, ha a származtatott asszociáció valamely más asszociációhoz tartozó kapcsolatoknak képezi a részét. A válogatást a megszorításon belül egy logikai kifejezésként adhatjuk meg.

Asszociációk értelmezése

Az asszociációk a modellezés különböző szintjein különböző módon értelmezhetők.

- Koncepcionális szinten az asszociációkkal az osztályokkal jelölt fogalmak közötti koncepcionális viszonyokat ábrázoljuk.
- A tervezés technikai aspektusából az asszociáció az elérés felelősségeként jelenik meg.
- A megvalósítás során az asszociáció az elérés megvalósítását jelenti.

elérés megvalósítása

Az elérés úgy biztosítható, ha az objektum rendelkezik a vele kapcsolatban álló másik objektum azonosítójával (OID), amin keresztül az objektumorientált környezet szolgáltatni tudja a megfelelő objektumot. A származtatott kapcsolat úgy értelmezhető, hogy az objektum közvetlenül nem rendelkezik a másik azonosítóval, de bizonyos műveletek eredményeként elő tudja azt állítani.

Az objektumazonosító lehet az objektumra mutató pointer, de lehet relációs adatbázisok esetén a másik tábla egyedi kulcsa. Ezért a másik objektum azonosítóját (például idegen kulcsot) nem szabad attribútumként felvenni, mivel a modellben az asszociációnak pontosan ez a jelentése.

asszociációk keresése

Az OO módszerek általában kevés segítséget adnak az asszociációk keresésére. Legtöbbször azt ajánlják, hogy az osztályokkal reprezentált fogalmak közötti kapcsolatokat vegyük fel asszociációkként. Sajnos, szinte mindegyik szakterületi osztály kapcsolatban van egymással, így az ajánlást követve könnyen pókhálóvá válhat a diagramunk.

A ***Q*** modellező módszer azt javasolja, hogy a jellegzetességek keresésével és a felelősségek megosztásával az asszociációk meghatározását is kapcsoljuk össze. Ha egy jellegzetesség megvalósításához igénybe vehetünk együttműködőt, akkor ellenőrizzük, hogy az elérhető-e a felhasználás (a „hívás") helyén. Ha nem, akkor strukturális módon kell biztosítanunk a kapcsolatot. A technika a navigációs irányt is meghatározza, mely a felhasználótól a használt felé mutat. Ha valamely más szituációban megfordul a felhasználás iránya, akkor az asszociációt módosítsuk kétirányúvá.

asszociáció felvétele

Minden új asszociáció felvételekor:

- Jelöljük be a navigáció irányát (nyilat húzzunk és ne vonalat).
- Határozzuk meg a számosságokat. Ha bizonytalanok vagyunk, akkor használjuk a tetszőleges (*) számosságot. Az 1 számosság elhagyható, az opcionális pedig 0-val rövidíthető.
- Jelöljük be, ha az asszociáció származtatott. A származtatás módját megszorításként adhatjuk meg.
- Nevezzük el az asszociációt, ha ezzel érthetőbbé válik a modellünk. Célszerű elnevezéseket használni, ha ugyanazon osztályok között több asszociáció is létezik.
- Nevezzük el a szerepeket, ha ezzel érthetőbbé válik a modellünk. Reflexív asszociációk esetén mindig használjunk legalább egy szerepnevet.

A módszer nem javasolja olyan szerepnév, vagy asszociációnév felvételét, amely valamelyik szerep osztályának az elnevezésével egyezik meg.

A *Q* módszer az asszociációk hagyományos értelmezését kiterjeszti és egzakt szemantikát határoz meg. Értelmezésében a kapcsolat az információs csomópontok között húzódó olyan út, amelyen az entitások információt közölhetnek egymással és így hathatnak egymásra, azaz befolyásolhatják egymás működését. Az egymással kapcsolatban lévő, önállósággal rendelkező elemek is „körberajzolhatók", mivel a kapcsolódó elemek bizonyos külső hatásokra együttesen reagálnak.

A szemlélet alapján a strukturális kapcsolat azt jelenti, hogy a kiindulási osztály objektumai a navigálhatóság irányának megfelelően megadják utat, azaz biztosítják az elérést. A megadás az attribútumokhoz hasonlóan műveletek segítségével történik.

A módszer a szerepekhez is javasolja az írhatóság, olvashatóság és módosíthatóság jelzését.

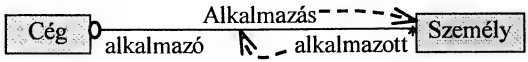
r◁	olvasható és módosítható, azaz elem felvihető és törölhető
r	csak olvasható, nem vihető fel és nem törölhető elem.
r◁◁	nem kérdezhető le
r+	nem törölhetők az elemek
r-	inicializálás után csak törölhetők az elemek, de nem vihetők fel
r◁+	az elem csak felvihető, és az elérése sem engedélyezett
r◁-	az elem csak törölhető, és az elérése sem engedélyezett

alapnév

Az elérést biztosító művelethez az alapnevet a következőképpen képezzük.

- Ha a kapcsolatban az elérni kívánt osztálynak van szerepneve, akkor az lesz az alapnév (például `alkalmazott`).
- Ha nincs szerepnév, de az asszociáció el van nevezve, akkor azt választjuk alapnévként (például `Alkalmazás`).
- Ha sem szerepnevet, sem asszociációnevet nem adtunk meg, akkor az alapnév az elérni kívánt osztály neve (például `Személy`).

85. Ábra: Alapnév kiválasztása



1 számosság

Ha az elérni kívánt ellentétes szerep 1 multiplicitású, akkor `alkalmazó` alapnév és `Cég` osztály esetén azt a következő attribútumként és műveletekként értelmezzük:

```
-alkalmazó: Cég                  -- ha nem származtatott
+getAlkalmazó(): Cég             -- ha olvasható
+setAlkalmazó(alkalmazó_: Cég)   -- ha írható
```

opcionális számosság

Opcionális esetben fel kell készülni arra, hogy a lekérdezés vagy egy üres azonosítót (pl. `null`) vagy egy logikailag üres objektumot ad.

```
-alkalmazó: Cég                  -- ha nem származtatott
+hasAlkalmazó(): Bool            -- ha olvasható
+getAlkalmazó(): Cég             -- ha olvasható
+setAlkalmazó(alkalmazó_: Cég)   -- módosítható/bővíthető
+clearAlkalmazó()                -- ha törölhető
```

Több számosságú szerep esetén az objektum több, azonos osztályhoz tartozó objektummal, azaz egy kollekcióval van kapcsolatban. A megvalósítás során ez azt jelenti, hogy a kollekciót egy megfelelő konténer osztályon keresztül érjük el, ami tartalmazza a kollekcióra vonatkozó attribútumokat és a kezelésükhöz szükséges műveleteket.

kollekciók

A *Q* módszer a kollekciók és a felsorolások definiálásra egy generikus szerkezetet ajánl, amelynek paraméterként megadhatjuk a tárolandó objektumok osztályát.

Egy halmaz (`Set`) kollekciótól általában a következő műveleteket várjuk el:

```
Set<X>:
  +/has(): Bool              -- van-e eleme
  +/has(x:X): Bool           -- van-e egy adott eleme
  +/every(): Every<X>        -- felsorolja az elemeit
  +add(x:X)+                 -- felvesz egy új elemet
  +remove(x:X)-              -- töröl egy elemet
  +removeAll()-              -- törli az összes elemet
```

Az `X`-et a tárolandó osztály nevével kell helyettesíteni. A kezdő / a lekérdezést (nem módosítást) jelöli, a művelet zárójele után megadható + és - pedig a felvitel, illetve törlés jellegű műveletre utal, melyeket az `{isQuery}`, `{add}`, illetve `{remove}` megszorításokkal is helyettesíthetünk.

felsorolások

A halmaz elemeit az `every` művelettel kérdezhetjük le, amely egymás után felsorolja a kollekció objektumait. Egy felsorolástól legtöbbször a következő műveleteket várjuk el:

```
Every<X>
    +/current(): X      -- megadja az aktuális elemet
    +/empty(): Bool     -- kiürült-e a felsorolás
    +next()             -- a következő elemre lép
```

A felsorolásosztály segítségével egy megfelelő ciklussal végigvehetjük a kollekció elemeit. A ciklus Java vagy C++ nyelven lehet például a következő:

```
for(Every x=s.every(); !x.empty(); x.next()) {
   x.current()...
}
```

több számosság

Az osztály több számosságú ellentétes szerepe így a következőképpen jelenhet meg:

```
-alkalmazott: Set<Személy>        -- ha nem származtatott
+getAlkalmazott(): Set<Személy>
```

Az `x` cég összes alkalmazottját például a következő módon sorolhatjuk fel:

```
x.getAlkalmazott().every()
```

publikálás

A *Q* módszer egy érdekes technikájának, a *publikálás* *(publishing)* segítségével tovább bővíthetjük az asszociációk szemantikáját. A publikálás generikus osztályok esetén azt jelenti, hogy annak jellegzetességeit „beemeljük" a használó osztályba.

Az `alkalmazott: Set<Személy>` módú felhasználás esetén például a halmaz `every` művelete `everyAlkalmazott(): Every<Személy>` módon a kiindulási osztályba is átkerül, amely a műveletet egyszerű delegációként továbbítja a kollekciónak.

Ha a `Cég` osztály az alkalmazottainak elérését a fent definiált `Set` tárolóosztály módosítható/olvasható használatával valósítja meg, akkor a publikálás technikájával a `Cég` a következő műveletekkel rendelkezik.

```
-alkalmazott :Set<Személy>
+getAlkalmazott() :Set<Személy>
+hasAlkalmazott() :Bool
+hasAlkalmazott(alkalmazott_: Személy) :Bool
+everyAlkalmazott(): Every<Személy>
+addAlkalmazott(alkalmazott_: Személy)
+removeAlkalmazott(alkalmazott_: Személy)
+removeAllAlkalmazott()
```

Java nyelven az alkalmazottak például a következő ciklussal vehetők sorra:

```
for(Every x=everyAlkalmazott(); !x.empty(); x.next() ) {
   ((Személy)x.current()).fizetésemelés();
}
```

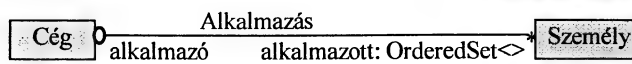
Ugyancsak a publikálás technikáját láthattuk korábban az egy számosságú, illetve opcionális szerepeknél, valamint az attribútumoknál(!) is.

szerep jellegének jelölése

A tároló osztály jellegét a szerep neve után, kettőspontot követően adhatjuk meg. Ha a kisebb és nagyobb jel között nem adunk meg paramétert, akkor oda az osztály nevét helyettesítjük.

Alapértelmezésként úgy tekintjük, hogy a több számosságú szerep és attribútum `Set` jellegű. A műveletek paramétere vagy visszatérési értéke esetén a * pedig az `Every` felsorolást jelenti. Mindkét generikus osztály műveletei felüldefiniálhatók, ha az alkalmazás ezt követeli meg.

86. Ábra: Tárolás jellege



Más jellegű tárolást további generikus szerkezetekkel definiálhatunk. Például egy `OrderedSet<X>` jelölheti azt a halmazt, melyben az értékek adott sorrendnek megfelelően helyezkednek el. A `Bag<X>` névvel olyan kollekciót vehetünk fel, amelyben egy objektum többször is előfordulhat. Ha az objektumok sorrendje lényeges, akkor definiáljunk egy `OrderedBag<X>` jelleget.

Hierarchikus adatszerkezetet jelölhetünk egy `Hierarchy<X>` elnevezésű generikus szerkezettel. A hierarchikus tárolás a hierarchia alsóbb szintjén álló objektu-

mok elérését is jelenti, ezért azt nem szükséges az alaposztályban külön, rekurzív kapcsolatként jelölni. Irányított körútmentes gráfként *(directed acyclic graph)* történő tárolást egy `Dag<X>` jelleggel adhatunk meg.

A kollekciókhoz hasonlóan az elemek közötti navigációt lehetővé tevő, az `Every`-nél bonyolultabb iterátokat is definiálhatunk.

A műveleteikkel megadott generikus tároló és a felsoroló osztályok megvalósítási módját, a műveletekhez tartozó konkrét utasításokat a fejlesztés későbbi szakaszában, a tervezés során határozzuk meg.

Elérési utak

Az asszociációk alapvető jelölései mellett az UML további technikákkal is rendelkezik. Minősítővel az objektumok egy adott érték alapján azonosíthatók. Asszociációs osztály segítségével az asszociációhoz attribútumokat és műveleteket is rendelhetünk. Aggregációs és kompozíciós kapcsolattal jelöljük a kapcsolódó objektumok élettartamának összefüggését.

Minősített asszociáció

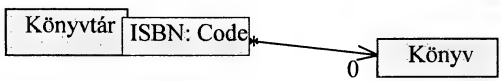
Több számosságú szerep esetén szükséges lehet, hogy a kollekción belül egyes objektumokat egy érték alapján egyedi módon azonosítsunk, esetleg adott értékhez meghatározzuk az objektumok egy csoportját. Ezt az érték-jellegű elérést, azonosíthatóságot a *minősítővel* `(qualifier)` adjuk meg, amely képletesen az ellentétes szerep multiplicitását csökkenti. Például egy könyvtárban egy könyvet az ISBN alapján közvetlenül is megtalálhatunk. Hasonlóan, a raktári száma alapján előkereshető egy könyvpéldány.

minősítő

kulcs jellegű elérés

A *minősített asszociáció* `(qualified association)` a kulcs jellegű elérés felelősségét jelenti.

87. Ábra: Minősítő



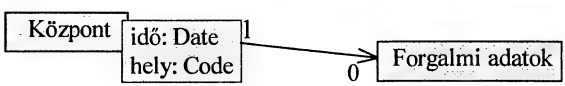
A minősítő egy olyan érték vagy értékkombináció, amely azonosítóként a kapcsolatban részt vevő objektumoknak egy részhalmazát, ideális esetben pedig pontosan egy objektumot választ ki. Egy fájlrendszerben ilyen azonosító például az állomány neve.

minősített asszociáció számossága

Az UML ajánlása szerint a minősített asszociációnak a minősítővel ellentétes oldalán mindig jelöljük a számosságot.

A ***Q*** modellező módszer minősítő esetén a leggyakrabban előforduló, opcionális számosságot tekinti alapértelmezettnek, melyet így nem szükséges kiírni. Ritkább esetben minden érték pontosan egy objektumot határoz meg — ekkor kiírjuk az 1 multiplicitást. Néha előfordulhat, hogy egy minősítőhöz több objektum is tartozik. Például egy fájlrendszerben a globális fájlnév-karakterekkel (pl.: `*.java`) több fájlra is hivatkozhatunk.

88. Ábra: Összetett minősítő



Az UML a minősítő értékeit attribútumokként tekinti, melyek neve mellett a típusa is megadható. Összetett minősítőt is felvehetünk, amelyek együttes értéke választja ki az objektumot vagy az objektumokat.

minősítők keresése

A szakterületi modellünk minősített asszociációit szintén a felhasználói felületek vizsgálatával találhatjuk meg. Olyan értékeket kell keresnünk, amelyek objektumokat azonosítanak.

minősítő és OID

A minősítő és az objektumazonosító (OID) között az a különbség, hogy a minősítő a felhasználói felületeken is megjelenő érték, míg az objektumazonosító rejtve marad, mivel az csak az OO környezet egy belső technikai megoldása.

minősítő értelmezése

A ***Q*** modellező módszer értelmezésében a minősítő a tárolóosztályban vagy publikálás esetén a kiindulási osztályban további metódusokként jelenik meg:

```
+hasKönyv(ISBN: Int): Bool           -- ha olvasható
+getKönyv(ISBN: Int): Könyv          -- ha olvasható
+setKönyv(ISBN: Int, könyv_: Könyv) -- ha új felvihető
+removeKönyv(ISBN: Int)              -- ha törölhető
+everyKönyvISBN(): *Int              -- ha olvasható
```

osztály neve

Esetenként a minősítő az ellentétes osztály attribútumaként is megjelenhet (ennek szükségessége is a felhasználói felületek vizsgálatából, illetve a felelősségek megosztásából határozható meg). Az osztály *nevének* nevezzük azt az attribútumát vagy attribútumait, amelyek értékei az osztály objektumait a teljes alkalmazásban egyedi módon azonosítják. A ***Q*** módszer javaslata alapján a név jellegű attribútumot a `{name}` megszorítással jelöljük, melyet úgy értelmezünk, hogy az osztályhoz kapcsolódó összes többi osztály a névnek megfelelő értékek megadásával egyedi módon el tudja érni az adott nevű objektumot, azaz mind rendelkezik a névnek megfelelő minősítővel (és így az ahhoz tartozó műveletekkel is).

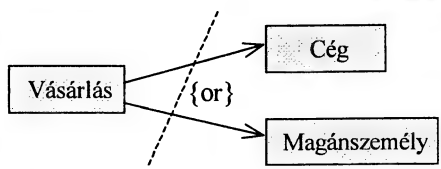
Az osztály nevét nem célszerű az elemzés kezdetén meghatározni. Általánosabb megoldást kapunk, ha azt először minősítőként modellezzük, mivel szükséges lehet, hogy egy osztály objektumait más asszociációk esetén más értékekkel azonosítsuk.

„Vagy" asszociáció

„vagy" asszociáció

„Vagy" asszociációval `(or-association)` jelöljük, ha egy konkrét objektum esetén több potenciális asszociáció közül egyidőben csak egy asszociáció realizálódhat. Másik megközelítésben az objektumhoz azonos asszociáción különböző osztályú objektumok is kapcsolódhatnak. A „vagy" asszociáció jele az `{or}` megszorítás.

89. Ábra: „Vagy" asszociáció

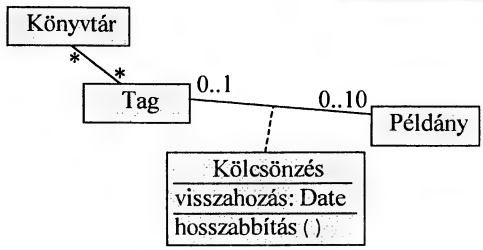


Asszociációs osztály

asszociációs osztály

A felhasználói felületek bizonyos értékeinél észrevehetjük, hogy azok nem egy osztályt, hanem valamely osztályok közötti asszociációt jellemeznek. Az *asszociációs osztályok* `(association class)` segítségével az asszociáció osztályként tekinthető, így ahhoz attribútumok, műveletek, illetve más asszociációk is rendelhetők.

90. Ábra: Asszociációs osztály



Az asszociációs osztály pontosan ugyanazt jelenti, mint az asszociáció, így az asszociációs osztály neve az asszociációnál is megadható.

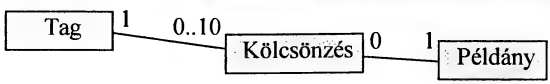
modell ellenőrzése

Gyakori hiba, hogy az asszociáció attribútumait és műveleteit az egyik (a több számosságú) osztálynál veszik fel. Példánkban a visszahozás dátumát és a hosszabbítás műveletét a példánynál is megadhattuk volna. Azt, hogy a jellegzetességek az osztályra vagy annak egy asszociációjára vonatkoznak, úgy ellenőrizhetjük, hogy gondolatban mindkét számosságot a legáltalánosabb „tetszőlegesre" (*) változtatjuk, majd sorra veszünk minden jellegzetességet. Ha egy példányt egyszerre többen is kölcsönözhetnek, akkor a visszahozás dátuma és a hosszabbítás esetén sem egyértelmű, hogy azok melyik `Tag` objektumra vonatkoznak. Ha ezt a két jellegzetességet a tagnál adnánk meg, akkor pedig az nem lenne egyértelmű, hogy azok mely példányra vonatkoznak.

Ha az asszociációban csak egyetlen szerep számossága nagyobb egynél, akkor az asszociációs osztály jellegzetességei a tervezés során a több számosságú osztályba csúsztathatók. Az elemzés során azonban célszerű az asszociációhoz tartozó jellegzetességeket a felvett asszociációs osztályban modellezni.

Ha a tervezés során az asszociáció jellegzetességeit nem olvasztottuk be a több számosságú szerep osztályába, akkor azt a megvalósításkor általában a kettő vagy több osztályt összekötő útvonalon helyezzük el.

91. Ábra: Asszociációs osztály megvalósítása



A modellezés során ez a jelölés azonban hiba, mivel lehetővé teszi, hogy egy `Tag` egy másik kölcsönzéssel ugyanazt a példányt kétszer is kikölcsönözze.

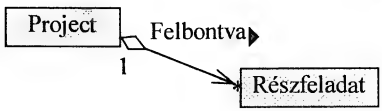
Aggregáció és kompozíció

Az *aggregáció* `(aggregation)` egy speciális asszociációt, a két elem közötti *egész-rész* `(whole/part)` viszonyt jelöli, például egy háznak a részei a falak, az ajtók és az ablakok. *aggregáció*

Az aggregáció az objektumorientált szemléletnek az egyik legtöbb vitát okozó technikája. Coad az aggregációra példaként a szervezet és hivatalnokai közötti viszonyt adta meg, míg mások szerint ez nem egész-rész viszony, mivel a hivatalnokok a szervezettől függetlenül is létezhetnek.

Az UML az aggregációs viszonyt az asszociáció „egész" szerepénél elhelyezett rombusszal jelöli. Képletesen: az „egész" magához vonzza az asszociációra utaló rombuszt. Az aggregáció az „egész" bizonyos dominanciáját jelenti a részek felett, ezért ternáris és magasabbrendű asszociációnál nem adható meg. *jelölés*

92. Ábra: Aggregáció



Az aggregáció az asszociáció különleges változata, melyet a következő jellemzők alapján azonosíthatunk:

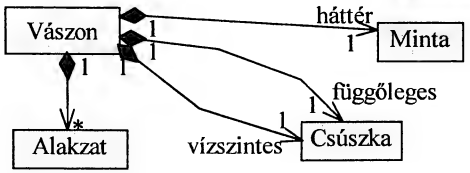
- Az aggregáció egy *„egész-rész" viszony*, így a rész logikailag az „egészen" belül helyezkedik el, illetve sok műveletben az egész a részeivel együtt vesz részt. *„egész-rész" viszony*
- Az aggregáció egy *antiszimmetrikus viszony*. Például egy szoba része lehet egy háznak, de a viszony megfordítása már nem igaz.
- Az aggregációs viszony *tranzitív*, így a ház szobájának ajtaja egyben a ház ajtaja is.
- Az aggregációra jellemző a *műveletek továbbítása, delegálása* `(delegation)`. Az egész a műveleteinek nagy részét úgy valósítja meg, hogy a felelősségeit áthárítja, megosztja a részeivel. Például egy ház kifestése valószínűleg az összes szoba kifestését is jelenti, vagy egy `Project` felfüggesztése az összes `Részfeladat` felfüggesztését is kiváltja. *műveletek delegációja*
- Az „egész" bizonyos attribútumait és kapcsolatait *felajánlja, propagálja* `(propagation)` a részeknek. Például a `Projectet` irányító vezető egyben a `Részfeladatok` vezetője is. Az értékek felajánlása vagy strukturális módon, egy visszafelé mutató kapcsolattal vagy a delegált műveletek paramétereként valósul meg. *attribútum-értékek propagálása*

A legtöbb aggregációra a *törlés továbbítása* `(cascading delete)` is jellemző, azaz az „egész" törlése a részek törlését is kiválthatja. Az UML azonban az aggregációt csak az egész-rész viszony jelzéseként javasolja, mindenféle kiegészítő szemantika nélkül. *törlés továbbítása*

kompozíció

Az értelmezéssel kapcsolatos viták feloldásaként az UML bevezette az aggregáció erősebb változatát, a *kompozíciót* `(composition)`, melyet az „egésznek" megfelelő szerepnél elhelyezett fekete rombusszal jelöl. A kompozíciós kapcsolat az „egész" lényegi alkotóelemeit jelöli, így a rész csak egyetlen „egész" objektumhoz tartozhat, illetve a rész az egésszel „együtt él és hal", azaz a rész élettartama az egészhez kötődik.

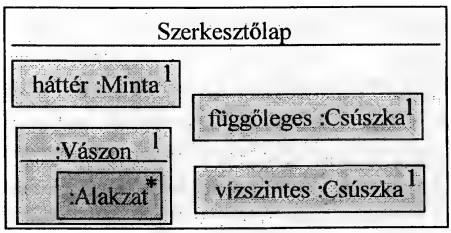
93. Ábra: Kompozíció



„beágyazott" jelölés

A kompozíció „beágyazott" módon is jelölhető. Ekkor a kompozit osztály neve alatt, attól egy vízszintes vonallal elválasztva grafikusan ábrázoljuk a tartalmazott komponenseket. Az alakzat névrészében a szerepnevet, majd kettőspont után az osztály nevét írjuk. A szerep multiplicitását az alakzat jobbfelső részében adjuk meg.

94. Ábra: Kompozíció „beágyazott" jelölése

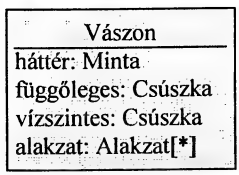


kompozíció és attribútum

A kompozícióra létezik egy attribútum-jellegű jelölés is. Ebben az esetben a szerepnevek lesznek az attribútumnevek, az osztályt az attribútum típusaként adjuk meg, melyet szögletes zárójelek között a számosság követhet. Ebben az esetben alapértelmezettként a „pontosan egy" számosságot tekintjük.

A ***Q*** módszer megengedi, hogy a [*] tetszőleges számosságot * módon jelöljük. A komponens és a kompozit szerepének a számosságát alapértelmezésként egynek tekinti az asszociációkhoz, illetve az attribútumokhoz hasonlóan.

95. Ábra: Kompozíció jelölése attribútumként



Erős és gyenge kapcsolat

erős és gyenge kapcsolat

A ***Q*** modellező módszer az objektumok közötti strukturális kapcsolatokat *erős* és *gyenge* kapcsolatként tipizálja. Értelmezésében, ha egy project objektumhoz erős kapcsolattal kapcsolódik egy részfeladat objektum, akkor a project törlése kiválthatja a részfeladat törlését is, ha arra erős kapcsolattal nem hivatkozik valamely más objektum, például egy másik projekt. Az erős kapcsolatnak egy speciális változata a *„vezérlő"* `(master)` kapcsolat. Ha a projekt vezérlő kapcsolattal hivatkozik a részfeladatra, akkor a projekt törlése akkor is kiváltja az alobjektum törlését, ha arra még más is hivatkozik. Objektumok közötti gyenge kapcsolat esetén egy objektum megszűnése nem hat ki a kapcsolódó objektumok élettartamára.

Ha egy megszüntetendő objektumhoz még mások is kapcsolódhatnak, akkor azoknak a megszüntetésről egy értesítést `(deleteOfX)` kell küldeni, és/vagy az objektumot, annak egy művelete segítségével logikailag töröltnek jelöljük.

A ***Q*** modellező módszer a kompozíciót a vezérlő kapcsolat, az aggregációt a többi erős kapcsolat, a jelöletlen asszociációt pedig a gyenge kapcsolat jelzésére használja.

A módszer úgy tekinti, hogy az attribútumként felvett, nem érték-jellegű (nem «value» vagy «enumeration») osztályok kompozíciós kapcsolatot jelölnek. A típus elé írt szimbólummal a kapcsolat jellegét érték-jellegű osztályra is megadhatjuk vagy módosíthatjuk:

- ◆ (szövegesen: <*>, vagy `{composition}` megszorítás) kompozíciós („vezérlő") kapcsolat; az alapobjektum megszűnésével a jellemző is megszűnik.
- ◊ (szövegesen: <>, vagy `{aggregation}` megszorítás) aggregációs kapcsolat; az alapobjektum megszűnésével a jellemző is megszűnhet, ha arra nincs több erős kapcsolat. A jellemzőhöz tartozó objektumot más is átveheti, így az az alapobjektum törlése után is megmaradhat.
- . (vagy `{link}`) asszociációs („gyenge") kapcsolat. Attribútumként megadott gyenge kapcsolat esetén alapértelmezettnek az opcionális számosságot tekintjük!

értékjellegű osztály kompozíciója

A módszer az *értékjellegű osztállyal* való kapcsolatok esetén további műveletekkel bővíti ki a szemantikát. Egy `a: ◊X` például a következőt jelenti (mivel nem adtunk meg kezdőértéket, de a számosság „1", ezért annak egy új objektumot kell értékül adnunk):

```
-a: X = new X()      -- ha nem származtatott
+getA(): X           -- ha olvasható
+theA(): X           -- ha olvasható
+setA(a_: X)         -- ha írható
+setTheA(a_: X)      -- ha írható
```

A `getA()` lemásolja az értéket, míg a `getTheA()` magát az objektumot adja vissza így az módosítható. Hasonlóan, a `setA()` átmásolja a változóba az értéket, a `setTheA()` azonban az objektumot cseréli le.

Attribútum és asszociáció

Az attribútumok nagyon hasonlítanak az asszociációkhoz. A modellezés során felvetődhet a kérdés, hogy egy osztály adott szempontú jellemzését egy attribútummal vagy egy hozzákapcsolt objektummal valósítsuk-e meg. Például egy `Számla` osztály esetén a `kelt` dátumát felvehetjük attribútumként. Másik megoldásként egy asszociációval is kapcsolódhatunk a Dátum osztályhoz.

96. Ábra: Attribútum vagy asszociáció

attribútum vagy asszociáció?

Az asszociációként történő modellezés az általánosabb eset, míg az attribútumként történő megadás több korlátozást is jelent. Egyrészt az attribútumként felvett jellemző egyirányú asszociációt jelöl, így nincs arra lehetőség, hogy egy adott dátum alapján közvetlenül megkapjuk az aznapi `Számlákat`. Asszociációként történő modellezés, illetve a megfelelő navigációs irány mellett fel tudjuk sorolni az adott naphoz tartozó számlákat (az asszociáció ugyanis részben ezt jelenti), de ha a `kelt` egy attribútum, akkor ezt csak közvetett módon, például a számlák sorravételével és ellenőrzésével tehetjük meg. Az attribútumként felvett objektumról az alapobjektumnak általában külön másolata van. A jellemző attribútumként való felvétele egy kompozíciós kapcsolatot is jelent, ezért a `Számla` megszűnésekor a `kelt` dátuma is törlődik.

Az attribútumként felvett jellemző felsorolt tulajdonságai mind az egyediség elvesztéséhez kapcsolódnak. Akkor célszerű tehát egy jellemzőt attribútumként és nem asszociációként felvenni, ha az alkalmazás szempontjából nem lényeges az egyedisége, hanem csak az értéke.

attribútumok minősítő jellegűek

Az attribútumértékek valójában minősítő-jellegűek. Például egy 1999-08-11 dátumértékhez egyértelműen hozzá tudjuk kapcsolni a megfelelő napot. Ezt a megfeleltetést esetenként csak a felhasználó végzi el, a felhasználói felületeken megjelenő értékek értelmezéseként vagy azok megadásaként. Az érték azonban az alkalmazásban is megjelenhet minősítőként. Az ilyen érték valójában a megfelelő osztály (esetünkben a dátum) *neve*, amely a teljes alkalmazásban egyedi módon azonosítja az osztály objektumait. A név hasonlít az objektumazonosítóhoz. Azonban az OID közvetlenül (szemantikusan) valósítja meg a hivatkozást, míg a minősítő és a név közvetve, „szintaktikusan", egy érték segítségével. Összefoglalva: az érték önmaga „neve", melyet annak értelmezésével a felhasználó is össze tud kapcsolni a valós világ megfelelő fogalmával.

Függőségek

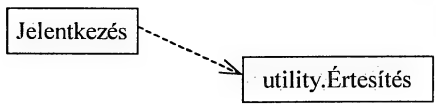
Egy osztály anélkül is felhasználhat egy másik osztályt, hogy azt egy asszociáción keresztül elérné. Az ilyen jellegű használatot az osztályok közötti *függőséggel* `(dependency)` ábrázoljuk. *függőség*

Az UML definíciója szerint egy modellelemtől *függ* egy másik modellelem, ha a definíciójának megváltozása a másik (a függő) elem megváltozását is eredményezheti. A függőség ebben az értelmezésben fordítási függőségként jelenik meg, így az elem definíciójának megváltozása a tőle függő többi modellelem változását, újrafordítását eredményezheti. *fordítási függőség*

Például két osztály között függőség van, ha ez egyik:

- üzenetet küld a másiknak,
- egy mezőjére hivatkozik,
- a másik osztály egy új objektumát készíti el,
- paraméterként felhasználja egy üzenetben, stb.

97. Ábra: Függőség



A függőség jellegét sztereotípiaként adhatjuk meg. Ime néhány az UML által definiált és eddig még nem ismertetett függőségek közül: *függőség jellege*

- «`use`» *(használat)*: a függő elem definíciója felhasználja a másik modellelem definícióját. Ez az alapértelmezett függőség, ezért csak akkor adjuk meg, ha hangsúlyozni szeretnénk, hogy nem más jellegű függőségről van szó.
- «`derive`» *(származtatás)*: a függő származtatott elem csak koncepcionális és az a használt konkrét vagy koncepcionális elemből számítható.
- «`friend`» *(„barát" kapcsolat)*: olyan speciális használati módot jelöl, amikor a függő elem jogosult elérni a használt elem védett `(protected)` és saját `(private)` jellegzetességeit is. Bizonyos OO nyelvek, például a C++, lehetővé teszik az elrejtett jellegzetességek „megnyitását" a barátként tekintett osztályok vagy műveletek felé.
- «`call`» *(„hívás")*: a függő elem meghívja a használt elemet vagy annak egy műveletét.
- «`instantiate`» *(„példányosítás")*: a függő osztály a használt osztálynak megfelelő típusú új objektumokat készít.
- «`instanceOf`» *(„példány")*: a függő elem (például objektum) a használt osztály konkrét példánya.
- «`powertype`» *(hatványtípus: a „típus-típusa")*: a matematikai hatványhalmaz koncepcióhoz hasonlóan *hatványtípusnak* `(power-type)` nevezzük azt a típust, amely példánya a (használt) alaptípus több objektuma. A hatványtípus bizonyos szempontból kollekció

jellegű. Alkalmas adatbázisok modellezésére, mivel egy adatbázistábla a rekordjának megfelelő alaptípus hatványtípusa, így a konkrét adatbázistábla az alaptípus rekordokként reprezentált objektumainak a halmaza.

- «refine» *(„finomítás")*: Finomítással jelölhetjük, hogy a függő elem azonos a „használt" elemmel, de annak több részletét ábrázolja. A finomítás egy pontosabb, kevésbé vázlatos képet határoz meg.
- «trace» *(„nyomkövetés")*: Nyomkövetéssel a fejlesztés során a modell kialakításának lépéseit ábrázoljuk. Például jelölhetjük, hogy egy, az elemzés során felvett használati esetet a tervezés fázisában csomagként valósítunk meg. Ekkor a csomagból a használati esetig vezető «trace» függőséget húzunk.

Függőséget az osztály jellegzetességéből, például egy műveletéből is indíthatunk, illetve egy jellegzetesség használatát is jelölhetjük.

Úgy tekintjük, hogy az azonos csomagban lévő osztályok használhatják egymás jellegzetességeit, így azok között sztereotípiával nem jelölt függőségeket általában nem ábrázolunk.

Általánosítás és pontosítás

Több osztályban ismétlődő közös jellegzetességek egyetlen általános osztályba emelhetők ki, melynek az eredeti osztályok a pontosításai, változatai lesznek. Az általános osztállyal a változatok absztrakt módon, jellegként kezelhetők. A megkülönböztetővel az általánosként tekintett alosztály esetén azonosítható, hogy melyik változathoz tartozik. A pontosítás történhet kiterjesztéssel, mely esetén az alosztályt újabb attribútumokkal és műveletekkel egészítjük ki, illetve egyes műveleteket átdefiniálhatunk.

A jellegzetességek összegyűjtése során észrevehetjük, hogy bizonyos jellegzetességek, felelősségek több osztályban is megismétlődnek. Ha találunk olyan osztályokat, amelyek közös attribútumokkal, műveletekkel és asszociációkkal rendelkeznek, akkor lehetséges, hogy az osztályokkal reprezentált fogalmaknak is van egy közös, általános fogalma. Ebben az esetben az ismétlődő jellegzetességeket kiemelhetjük egy olyan osztályba, amelyet az általános fogalomra utaló névvel nevezünk el, és egy speciális módon összekapcsoljuk a típust és altípusait.

általános fogalom

típus

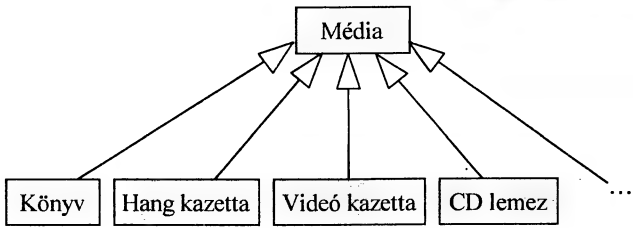
A fogalmak között a valós világban általában sokkal több általánosítás-pontosítás kapcsolat van, mint amennyit az alkalmazásban megjelenítünk. Akkor célszerű jelölni a fogalmak általános változatát, ha több lehetséges változatra bizonyos esetekben jellegként szeretnénk hivatkozni. Egy könyvtári rendszerben a `Könyv`, a `Videó kazetta`, stb. osztályok esetenként egy általános `Médiaként` jelennek meg, amelyeket azonos módon használunk, például mindegyik kölcsönözhető. Azonban az alkalmazásban nem feltétlenül jelennek meg a `Szöveges média`, `Hang média`, stb. közbülső jellegek. Ha több, különböző fogalmat bizonyos esetekben jellegként használunk, akkor a közös jellegzetességeket kiemelhetjük egy általánosított osztályba.

Általánosítás és pontosítás

Az *általánosítás* `(generalization)` vagy más néven *öröklés* `(inheritance)` egy viszony az *ős-* vagy *szuperosztály* `(ancestor, superclass)` és annak egy vagy több pontosított változata, a *leszármazottak* `(descendent)` vagy *alosztályok* `(subclasses)` között.

általánosítás, öröklés

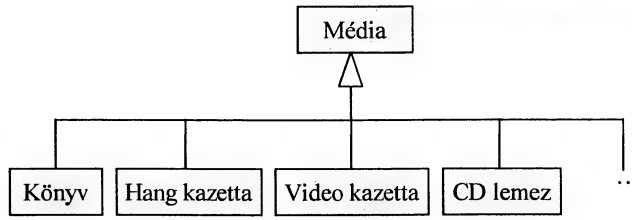
98. Ábra: Általánosítás



jelölés

Az általánosítás viszonyt az UML az alosztályból az ősosztályba vezető, háromszögben végződő vonallal jelöli. A három ponttal hangsúlyozhatjuk, hogy a modellben még léteznek további változatok, de azokat nem ábrázoljuk a diagramon. Az általánosításnak létezik egy „fa" jellegű, leágazásos jelölése is.

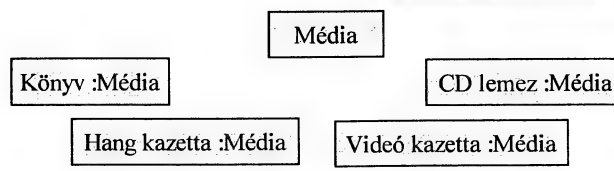
99. Ábra: Általánosítás „fa" jelölése



Az általánosítás viszonyhoz sztereotípiát és megszorítást is megadhatunk. „Fa" jelölés esetén ezt az elágazás közelébe helyezzük, vagy a különálló jelöléseket egy szaggatott vonallal áthúzzuk és a megszorítást a vonal mellé írjuk.

Q modellező módszer esetén a pontosítást szöveges módon is jelölhetjük. Ekkor az elem típusát az elnevezést követő kettőspont után írjuk. Például `Könyv :Média` (kiolvasva: a Könyv egy *(„is a")* Média). A jelölés összhangban van egyes programozási nyelvekkel (pl. a C++-al), sőt, az UML más jelöléseivel is.

100. Ábra: Szöveges jelölés (Q)



{incomplete}

{complete}

„..."

Az `{incomplete}` („nem teljes") megszorítással jelöljük, hogy az ősosztálynak létezhet még a modellen kívüli változata is. Ellentéte a `{complete}` megszorítású *teljes általánosítás*, mely esetén minden, ősosztályként tekintett objektum a modellben lévő (bár nem feltétlenül ábrázolt) valamely alosztály objektuma. Az `{incomplete}` nem azonos a három pont (...) jelölésével, mivel ez utóbbi mindössze arra utal, hogy a modellben létezik még több változat, de azt nem ábrázoltuk a diagramon.

Az általánosítás-pontosítás viszony koncepcionális szinten az általános jelleg és annak speciális változata közötti kapcsolatot ír le. Technikai szempontból a viszony úgy jelenik meg, hogy az ősosztály jellegzetességeivel (attribútumaival, műveleteivel, asszociációival és függőségeivel) annak minden alosztálya is rendelkezik. A megvalósítás oldaláról az általánosítás a jellegzetességek átvételének informatikai megoldását jelenti, amelyet általában örökléssel *(inheritance)*, vagy esetenként delegációval oldanak meg.

Az ősosztály egy jelleget fejez ki, ezért ahol a „jelleget", az ősosztályt használjuk fel, ott bármely alosztályát is megadhatjuk. Például, ha a `Médiák` kölcsönözhetők, akkor egy `Könyv` is, amely egy speciális `Média`. Összefoglalva: az alosztály objektumát *behelyettesíthetjük* *(`substitutability`)* minden olyan helyre, ahová valamely ősosztályának az objektumát várjuk. A helyettesíthetőség akkor valósítható meg, ha az alosztály *összhangban van* *(`conform`)* az ősosztályával, így az ősosztály minden jellegzetességével is rendelkezik.

helyettesíthetőség, „összhangban van"

A C++ programozási nyelv `private`-öröklés technikája alapján az UML definiál egy `«implementation»` sztereotípiájú pontosítást is, amely az ősosztály publikus attribútumait és műveleteit saját `(private)` jellegzetességekként veszi át. Az öröklésnek ez a fajtája azonban megsérti a helyettesíthetőség elvét, mivel az alosztálynak az ősosztályként tekintett objektuma esetén az elrejtett jellegzetességek elérhetővé válnak, ezért ezt nem is tanácsos használni. Az `«implementation»` pontosítás esetén az alosztály felhasználja vagy megvalósítja az ősosztály jellegzetességeit, de a külvilág elől el is rejti azt. A technika valójában egy specializálható rejtett (`private`) osztály komponensként felhasználást rövidíti, ezért azt célszerűbb kompozíciós viszonyként megadni.

private öröklés

Az általánosítás egyben a kiemelés, az absztrakció lehetősége. Modellünkben a közös jellegzetességekkel rendelkező fogalmaknak általában a valós világban is egy absztraktabb fogalom, egy jelleg feleltethető meg. A kiemeléssel a hasonló jelleget nem külön, minden egyes változatban, hanem egyetlen helyen, az ősosztályban adhatjuk meg. A modellünk így kisebb méretű, és kezelhetőbb lesz, valamint a konzisztenciája is könnyebben fenntartható. Ellentétes oldalról a kiemelés az *újrafelhasználás* *(`reuse`)* lehetőségét jelenti, így az egyszer már megfogalmazott általános jellegű osztály az aktuális követelményeknek megfelelően specializálható.

kiemelés lehetősége

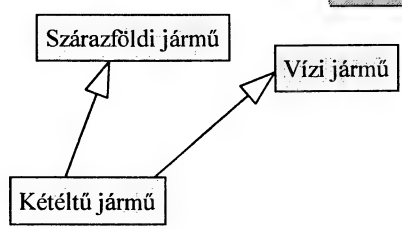
újrafelhasználás

Többszörös osztályozás

A valóságban egy fogalom természetesen több általános jelleggel is rendelkezhet. A klasszikus példa szerint (amelyet például az OMT könyvében is megtalálhatunk) egy `Kétéltű jármű` egyszerre `Szárazföldi jármű` és `Vízi jármű` is. Ha a modellünkben egy osztály több ősosztálynak is a pontosítása, akkor ez az *egyesített osztálynak* *(`join class`)* nevezett alosztály örökli minden egyes ősének a jellegzetességeit.

egyesített osztály

101. Ábra: Egyesített osztály



egyszeres és többszörös osztályozás

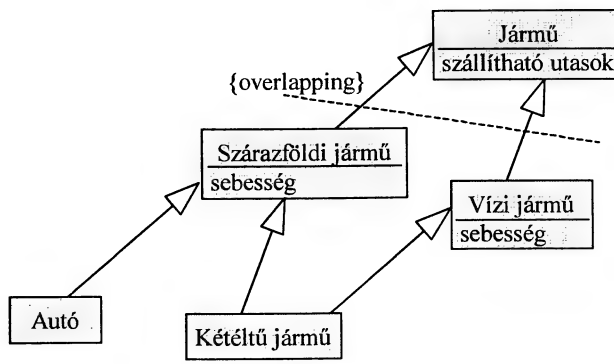
A technikát *többszörös osztályozásnak* (`multiple classification`) nevezik, mivel ekkor az egyesített osztály objektuma több olyan ősosztályként is megjelenhet, amelyek nincsenek feltétlenül általánosítás viszonyban (a `Szárazföldi jármű` nem pontosítása a `Vízi járműnek`, sem fordítva). Ennek ellentéte az *egyetlen osztályozás* (`single classification`), amikor az objektum pontosan egy osztályhoz, illetve annak ősosztályaihoz tartozik. Többszörös osztályozás esetén célszerű átgondolni, hogy melyek a megengedett kombinációk.

A *Q* módszer szöveges jelölése esetén az osztály több ősosztályát a kettőspontot követően, vesszővel elválasztva sorolhatjuk fel: `Kétéltű jármű :Szárazföldi jármű, Vízi jármű`.

Az egyesített osztály konkrét objektumát minden egyes ősosztály példányaként is el kell érnünk, azaz minden olyan helyen fel kell tudnunk használni, ahol az általános jelleget jelentő ősosztály is megjelenhet. Az elv azonban nagyon nehezen valósítható meg a számítógépes implementáció során, ezért ezt bizonyos programozási nyelvek nem is teszik lehetővé.

Többszörös osztályozás esetén újabb problémát jelent, ha bizonyos ősosztályok egy közös ősnek a pontosításai, például a `Szárazföldi jármű` és a `Vízi jármű` is egyaránt `Jármű`. Ha a `Szárazföldi jármű` és a `Vízi jármű` egyaránt rendelkezik egy `sebesség` attribútummal, akkor az egyesített osztálynak lesz egy szárazföldi és egy vízi járműhöz tartozó sebessége is. A kérdés az, hogy a közös ősosztály jellegzetességei, például a `szállítható utasok` nevű attribútuma azonos vagy külön jellegzetességként jelenik meg az egyesített osztályban.

102. Ábra: Átfedő pontosítás



kizáró és átfedő általánosítás

Az UML az általánosítás viszonyokat két csoportba sorolja. A *kizáró* vagy *diszjunkt* (`disjoint`) általánosítás esetén az ősosztály egy objektuma egyszerre csak egy alosztályhoz tartozhat. Ha az általánosítás *átfedő* (`overlapping`), akkor az objektum egyszerre több alosztály példánya is lehet. A két típust az általánosításokra vonatkozó `{disjoint}` vagy `{overlapping}` megszorítással jelöljük.

Úgy tekintjük, hogy átfedő általánosítás esetén mindkét alosztálynál ugyanarról az ősosztályról van szó, ezért az egyesített osztályba az ősosztály jellegzetességei csak egyszer kerülnek be, így a példánkban a `Kétéltű járműnek` csak egy `szállítható utasok` attribútuma lesz. Az `{overlapping}` megszorítással jelölt szaggatott vonal olyan határnak tekinthető, mely „felett" a jellegzetességek csak egyszer kerülnek be a leszármazottakba, még ha az ősosztályt több útvonalon keresztül is érik el. Kizáró általánosítás esetén a több úton is elért ősosztályt különbözőeknek tekintjük, mivel egy objektum nem lehet egyszerre két alosztályának is a példánya, ezért az ott lévő jellegzetességek az egyesített osztályban megismétlődnek. Ha példánkban az `{overlapping}` általánosítást kizáróra változtatnánk, akkor az azt jelentené, hogy a `Kétéltű járműnek` két, külön szárazföldi jármű és külön vízi jármű esetén érvényes `szállítható utasok` attribútuma van.

A kizáró általánosítást általában statikus osztályozás esetén használjuk, amikor az objektum nem változtathatja meg a típusát. Átfedő általánosítást pedig általában dinamikus osztályozás esetén használunk, amellyel az objektum olyan lehetséges típusait jelöljük, amelyek az időben változhatnak.

Megkülönböztető

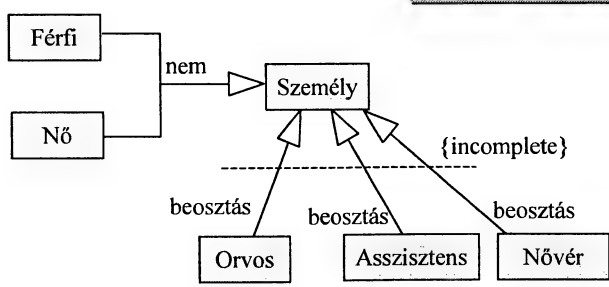
A *megkülönböztető* `(discriminator)` a pontosítást egy szempont megadásával elnevezi. Az azonos nevű megkülönböztetőkkel így az általános elem változatait csoportosíthatjuk.

megkülönböztető

A megkülönböztetőt az általánosítást jelölő, háromszögben végződő vonal elnevezéseként adjuk meg vagy az elágazásos „fa" jelölés esetén a csomóponthoz írjuk.

Az azonos megkülönböztetővel összekapcsolt altípusokat általában (de nem feltétlenül) kizáróknak tekintjük.

103. Ábra: Megkülönböztető



A ***Q*** modellező módszer az OMT technikáját átvéve a megkülönböztetőt olyan speciális felelősségként tekinti, mellyel az ősosztályként tekintett objektumról is meg tudjuk állapítani, hogy az mely altípus, mely alosztály eleme. Például egy `Személyként` tekintett objektum ismeri a beosztását, illetve meg tudja magáról mondani, hogy a `nem` szempontjából `Férfi` vagy `Nő`. Az egyes alváltozatok természetesen különböző jellegzetességekkel is rendelkezhetnek, például a `Nő` osztálynak létezhet `leánykori név` attribútuma.

megkülönböztető értelmezése

A *Q* módszer értelmezésében a típus azonosítása a következőképpen történik. A `d` megkülönböztetővel jelölt pontosítás esetén az ősosztálynak (a `Személynek`) létezik a következő művelete

```
+hasD(className: DName): Bool
```

valamint minden alosztályban létezik egy `DName` típusú (mely általában elemi típus, de lehet `String` is) `className` nevű konstans. Egy `Személy` objektumról így a következő módon kérdezhető le, hogy `Orvos` beosztású-e:

```
+hasBeosztás(Orvos.className)
```

Ha szükséges, akkor kiegészítésként az ősosztályban alosztályonként egy származtatott logikai értékű attribútummal is megvalósíthatjuk a lekérdezést:

```
+/orvos: Bool
```

amely a következő műveletként jelenik meg:

```
+isOrvos(): Bool
```

Kizáró `{disjoint}` pontosítás esetén a megkülönböztető az ősosztály (felsorolásos) attribútumaként is implementálható, amelyet az alosztályok az elkészítésükkor állítanak be. A felsorolás elemei az alosztályok neveinek felelnek meg.

A Java nyelvben az `o instanceof C` módon minden objektumról lekérdezhető, hogy az egy adott osztály példánya-e. A *Q* módszer megoldása azonban azt is lehetővé teszi, hogy átfedő pontosítás esetén is azonosíthassuk a változatot, még akkor is, ha a pontosítás hierarchikus, vagy ha az osztály dinamikusan változtatja a típusát.

Minden alosztály ismeri az ősosztályait, a típusait. A megkülönböztető megadását ellentétes irányú navigációként értelmezzük, mivel az ősosztályként tekintett objektumról megállapíthatjuk, hogy az melyik alosztállyal jelölt változatba tartozik.

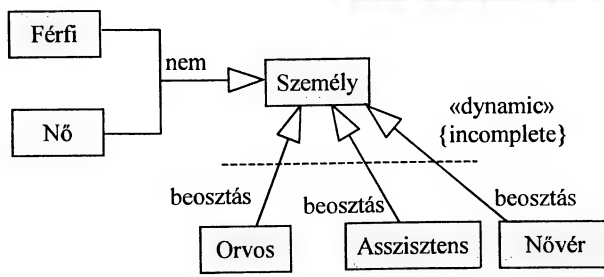
Dinamikus és többszörös változat

dinamikus osztályozás

Statikus osztályozásnak `(static classification)` nevezzük azt az esetet, ha az objektumnak a teljes élettartama alatt nem változik meg a típusa. *Dinamikus osztályozás* `(dynamic classification)` esetén azonban megengedjük, hogy az objektum altípusa megváltozhasson.

A dinamikus osztályozást Fowler és Scott alapján a megkülönböztetőhöz írt «`dynamic`» sztereotípiával jelölhetjük.

104. Ábra: Dinamikus osztályozás



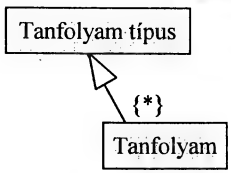
A modellezés során gyakran előfordul, hogy megjelenik egy olyan típus, mely értékeihez több konkrét objektum is tartozik. Korábbi példánkban ilyen volt a `Tanfolyam típus` és a `Tanfolyam`, mivel a `Tanfolyam típus` által meghatározott értékekhez több meghirdetett `Tanfolyam` is tartozhat.

Itt tehát típus-altípus kapcsolat van, de ez nem modellezhető a pontosítás hagyományos módján, mivel az alosztályok az ősosztálynak nem csak a jellegzetességein, hanem annak értékein is osztoznak. Ezt az esetet úgy hasonlíthatjuk az asszociációhoz, mintha az ős egy objektumához a változat több objektuma is tartozhatna.

A *Q* módszernek ezt a technikáját, a *több számosságú változatot* az altípushoz írt `{*}` megszorítással jelölhetjük, mivel pontosítás esetén nem adható meg számosság.

több számosságú változat

105. Ábra: Több számosságú változat



A dinamikus osztályozás egy egyszerű megvalósítási módja, ha a pontosítást asszociációként vesszük fel, a változatokban pedig az ős jellegzetességeit is elérhetővé tesszük.

Kiterjesztés és korlátozás

A személet szintjén a pontosítás mindig az általános lehetséges értékeinek egy részhalmazát jelöli. Az általános elem vázlatosabban, kevesebb jellegzetességgel határozza meg a valóságnak megfelelő fogalmat, így az ősosztályhoz mindig több entitás vagy érték tartozhat, mint a speciálisabb változathoz.

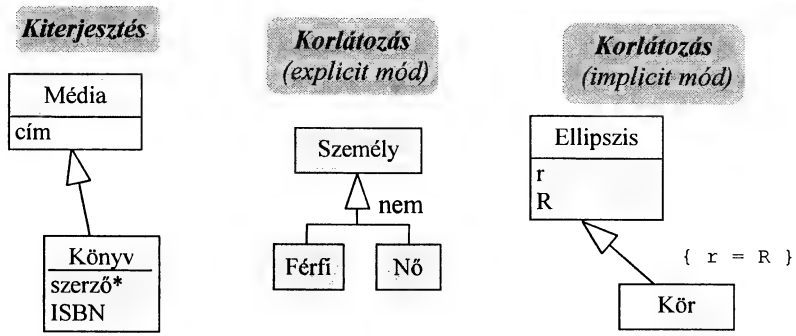
Egy osztály pontosítása kiterjesztéssel és korlátozással is történhet. *Kiterjesztés* `(extension)` esetén az ősosztályt új jellegzetességekkel, attribútumokkal, asszociációkkal vagy műveletekkel egészítjük ki. Az új műveletek általában az új attribútumokat kezelik. *Korlátozás* `(restriction)` esetén a pontosított alosztály az ős attribútumai által felvehető értékeket korlátozhatja.

kiterjesztés és korlátozás

A klasszikus példa szerint az `Ellipszis` osztály korlátozással `Kör` osztállyá pontosítható: ekkor az ellipszis kis- és nagytengelye azonos hosszúságú. Hiba lenne a `Kör` osztályt `Ellipszis` osztállyá kiterjeszteni, mivel az ellipszisek halmaza a körök halmazát is magában foglalja. Korlátozás esetén egy művelet (például a kör hossztengely menti nyújtása) kivezethet az alosztályból, de az objektum az ősosztálynak még ekkor is eleme marad.

Korlátozás esetén a pontosítást jelölő vonal mellé írt megszorítással adhatjuk meg az attribútumok értékeinek korlátozását.

106. Ábra: Kiterjesztés és korlátozás



A korlátozás tekinthető az ősosztály (példánkban az `Ellipszis`) olyan felelősségeként, mellyel meg tudja állapítani magáról, hogy teljesül-e rá a megszorítás, ezért egyben a vizsgált alosztály objektuma is.

A megkülönböztető tekinthető egy speciális (explicit) korlátozásnak, mivel ekkor az altípusba tartozást nem az ős attribútumain értelmezett *szabály* (logikai kifejezés), hanem egy *érték* alapján, általában annak egyszerű egyenlőségvizsgálatával hajtjuk végre.

korlátozás értelmezése

A ***Q*** módszer a korlátozással származtatott D alosztály esetén az ősosztály következő műveletét rendeli:

```
+/isD(): Bool
```

A megszorítás pontosan a tesztelendő logikai kifejezést határozza meg. A korlátozás azonban egy feltételes delegációt is jelent. Példánkban, ha a `Kör` az `Ellipszis` bizonyos műveleteit átdefiniálja és a művelet meghívásakor az ellipszisként tekintett objektum éppen kör, akkor a `Kör` osztály műveletét kell végrehajtani.

A kiterjesztés és korlátozás nem egymás ellentétei, így lehetséges, hogy egy pontosítás korlátozza az ős attribútumait, ugyanakkor azt újabb jellegzetességekkel is kiegészíti. Ez az eset azonban mindig átírható egy korlátozással képzett altípus további kiterjesztéseként.

Műveletek átdefiniálása

A pontosítás során az alosztály kiegészítheti az ősosztály jellegzetességeit, vagy átdefiniálhatja azokat. Az OO környezetek általában nem engedélyezik az ősosztály attribútumainak átdefiniálását, mivel az alosztály ősosztályként tekintett objektumában azok már léteznek.

A ***Q*** modellező módszer az attribútumokat műveleteken keresztül éri el, így itt az attribútumok átdefiniálása is lehetséges.

A műveletek *felüldefiniálását* (`overriding`) az objektumorientált nyelv megfelelő mechanizmusa, a *kötés* (`binding`) teszi lehetővé, amely minden objektum esetén megadja az adott művelethez tartozó megfelelő működést, a metódust.

műveletek felüldefiniálása, kötés

Az ősosztály műveleteit több okból is felüldefiniálhatjuk, például:

- Az alosztály kiegészítő jellegzetességei miatt a *műveletet* is *kiterjeszthetjük*, a működéséhez bizonyos többletet adhatunk.
- Az általánosabb ősosztályban a műveletnek csak egy *alapértelmezett* változatát adjuk meg, melyet az alosztályban *konkretizálhatunk*.
- Korlátozás esetén meg akarjuk *tiltani* a művelet bizonyos változatait.
- A pontosított alosztályban az objektumról több információ áll rendelkezésre, így az alosztály bizonyos műveleteket *hatékonyabban* hajthat végre.

Az alosztály felüldefiniálhatja az ős bizonyos műveleteit, ugyanakkor az alosztály objektuma az ősosztály objektumaként is szerepelhet (egy `Férfi` objektum tekinthető `Személyként` is). Ez azt jelenti, hogy ugyanahhoz a művelethez más objektum esetén más működés is tartozhat, ha az más alosztály példánya. Például egy `Alakzat` osztály `kirajzolás` művelete másképp jelenhet meg `Téglalap` és `Ellipszis` esetén. A technikát *polimorfizmusnak*, *többalakúságnak* (`polymorphism`) nevezzük, amivel arra utalunk, hogy egy általánosként tekintett objektum (például egy `Személy`) megjelenhet több változatban, alakban is (például `Férfi` vagy `Nő` objektumként), mely változatok az ősosztályban az eltérő működéssel figyelhetők meg.

polimorfizmus

Sajnos az UML és általában az OO programozási nyelvek nem jelölik, hogy a felvett művelet egy új, vagy azzal egy már létezőt definiálunk át.

A ***Q*** módszer a művelet felüldefiniálásának jelölésére az `{override}` megszorítást ajánlja.

```
fn() {override}
```

Ekkor visszatérési értékként alapértelmezettnek az ős azonos műveletének visszatérési értékét tekintjük.

A módszer definiálja a műveletek pontosítását, kiterjesztését is, melyet a kiterjesztett műveletek felsorolásával adunk meg. A művelet visszatérési értékének típusa a visszatérési értékekből képzett összetett típus.

```
fn() :a(), b(), c()
```

Ha az ősosztály azonos nevű műveletét terjesztjük ki, akkor rövidítésként az `{extend}` megszorítást használjuk.

```
fn() {extend}
```

A művelet kiterjesztése azt jelenti, hogy először sorrendben a kiterjesztendő műveletek hajtódnak végre, majd csak ezután a módszer saját működése.

Azonos művelet kiterjesztése vagy átdefiniálása esetén ellenőriznünk kell, hogy az ősosztályban valóban létezik ilyen szignatúrájú művelet, ellenkező esetben pedig azt, hogy valóban új műveletet definiálunk, nem pedig egy már létezőt írunk felül.

Osztályhierarchia

fa, vagy irányított körútmentes gráf

Egyetlen osztály sem lehet önmagának az ősosztálya, ezért az általánosítás-pontosítás viszonyokban nem lehet körút. Ha az OO környezet nem enged meg többszörös osztályozást, akkor az öröklési viszonyok egy-egy „faként" megjeleníthető hierarchiát alkotnak. Bizonyos programozási nyelvek azt is kikötik, hogy a hierarchia legáltalánosabb eleme, a fa „gyökere" csak a rendszer egyetlen (például `Object` elnevezésű) osztálya lehet.

Többszörös osztályozás esetén az öröklési viszonyok egy irányított körútmentes gráfot *(dag: directed acyclic graph)* alkotnak, mely minden pontjának több őse és több leszármazottja is lehet, de csak az ősök vagy csak a leszármazottak felé haladva nem lehet körút, nem juthatunk vissza a kiindulási pontra.

{root} {leaf}

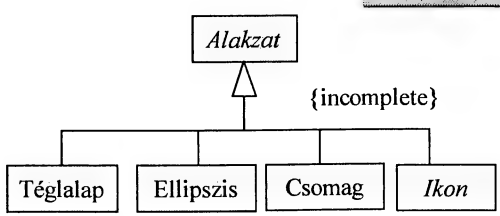
Az UML az öröklési gráf legáltalánosabb „gyökér" elemének vagy elemeinek jelölésére definiálja a `{root}` megszorítást. A tovább már nem pontosított osztályok, a fa „levélelemei" esetén a `{leaf}` megszorítást használhatjuk. A `{leaf}` jelölésével ugyanakkor meg is tiltjuk, hogy az osztálynak további változatait definiáljuk. Java nyelv esetén ezért a `{leaf}` az osztálynál megadott `final` kulcsszóval egyezik meg.

A `{leaf}` megszorítás műveletek esetén is megadható, mely ebben az esetben azt jelenti, hogy az alosztályban a művelet nem definiálható át (és nem is terjeszthető ki), ezért az alosztályban nem adhatunk meg az eredeti szignatúrával azonos műveletet. Ha az osztály pontosítását tiltjuk meg, akkor ez egyben azt is jelenti, hogy egyetlen művelete sem definiálható át.

Absztrakt osztályok és interfészek

A közös jellegzetességekkel rendelkező osztályok általánosítása esetén megjelenhet olyan általános osztály, amelyhez már közvetlenül nem, csak a leszármazottakon keresztül tartoznak objektumok. A klasszikus példa szerint egy grafikus rendszerben egy `Vásznon` megjeleníthetünk `Alakzatokat` (tehát szükség van a kiemelésre), de az alkalmazásban nem készíthetünk `Alakzat` objektumot, csak egy konkrét alakzatot, például `Téglalapot` vagy `Ellipszist`.

107. Ábra: Absztrakt osztály



absztrakt és konkrét osztály

A csupán lehetőségként létező osztályt *absztrakt osztálynak* `(abstract class)` nevezzük, mely egy olyan keretet határoz meg, amely csak valamely specializált változataként jelenhet meg. Az absztrakt osztálynak ezért nem készíthetők objektumai. Ellentétét *konkrét osztálynak* `(concrete class)` nevezzük, amelyből közvetlenül is készíthetők objektumok.

{abstract}

Az UML az absztrakt osztályt az osztály elnevezése mögé írt `{abstract}` megszorítással jelöli, vagy a nevet dőlt betűvel jeleníti meg.

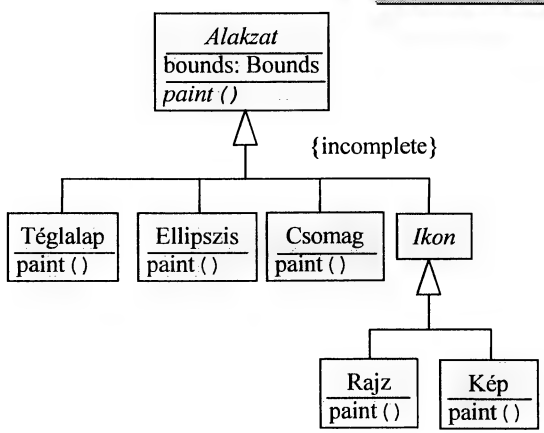
Az absztrakt osztály bizonyos műveletek esetén gyakran csak egy alapértelmezett tevékenységsorozatot határoz meg. Ha például az `Alakzatnak` létezik egy `contains(point: Location): Bool` lekérdezése, amely választ ad arra a kérdésre, hogy egy pont az alakzaton belül helyezkedik-e el, akkor az alapértelmezett működés lehet annak az ellenőrzése, hogy a pont az alakzat befoglaló téglalapján `(bounds)` belül van-e.

absztrakt műveletek

Az is lehetséges, hogy az absztrakt osztály mindössze egy adott szignatúrájú művelet létezését jelenti ki és ahhoz még alapértelmezett működést sem ad meg. Ha az alosztály ezeket a működés (metódus) nélküli üres, *absztrakt műveleteket* `(abstract operation)` nem definiálja felül, akkor az osztály továbbra is absztrakt marad. Konkrét alosztálynak ezért kötelező felüldefiniálni ezeket a műveleteket.

Az absztrakt műveletet dőlt betűvel vagy az `{abstract}` megszorítással jelöljük.

108. Ábra: Absztrakt művelet

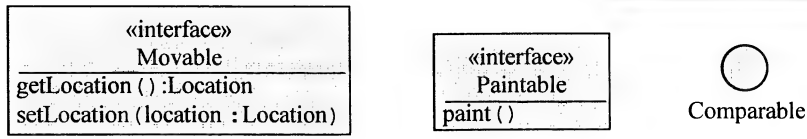


interfész

Az OO modellezés szóhasználatában felületnek vagy *interfésznek* `(interface)` nevezzük azt a speciális absztrakt osztályt, amely kizárólag látható (publikus) absztrakt műveleteket tartalmaz. Az interfész így attribútumokkal nem rendelkezhet (legfeljebb konstansokkal), mindössze bizonyos műveletek létezését jelenti ki, ezért az alosztályokkal szemben támasztott követelményekként, azok elérésének lehetőségeként jelenik meg.

Az UML az interfészt külön, osztály-jellegű, de mégsem osztály modellelemnek tekinti, melyet az «interface» sztereotípiával jelöl. A sztereotípiához a kör ikont rendeli, így az interfészt vázlatosan a kör alá írt megnevezésként is megjeleníthetjük.

109. Ábra: Interfész



lehetséges használati módok

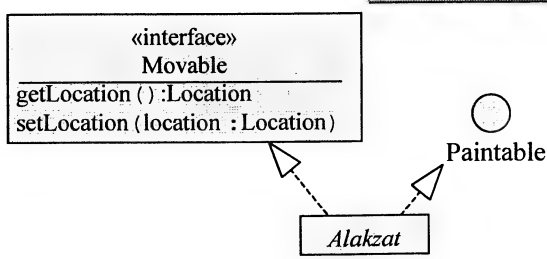
A ***Q*** modellező módszer az interfészeket lehetőségekként, az elérés lehetséges módjaiként értelmezi. Az interfészeket ezért gyakran a lehetőségre utaló névvel (`Movable`: mozgatható, `Paintable`: kirajzolható, `Comparable`: összehasonlítható) nevezzük el. Mivel az objektumokat funkcionális csomópontokként értelmezi, ezért elvileg megengedi, hogy az interfészek is rendelkezzenek attribútumokkal és asszociációkkal, melyek a megfelelő kezelőfüggvények (`setA`, `getA`, stb.) létezését rövidítik. Ez a megközelítés azonban nem fér be a szabvány kereteibe.

Az interfésznek is lehet általánosítása és pontosítása, mellyel a közös jellemzők létezését kiemelhetjük vagy újabbak létezését követelhetjük meg. Ha egy osztály *megvalósítja* *(`implements`)* az interfészben felsorolt műveleteket, akkor azt az UML az osztály és interfésze közötti speciális viszonnyal, a *realizációval* *(`realize`)* jelöli. A realizáció azt jelenti, hogy az osztály az interfésznél megadott összes jellegzetességgel rendelkezik.

realizáció

A realizációt az UML a megvalósítótól a megvalósított elemig vezető, háromszögben végződő szaggatott vonallal jelöli.

110. Ábra: Interfész és realizáció



interfészek használata

Ha az osztály megvalósít egy interfészt, akkor az osztály objektumai elérhetők az interfész által megkövetelt módon. Az interfészek alkalmasak arra, hogy használati módokat határozzunk meg. Készíthetünk például egy olyan műveletet, amely a vásznon az egér segítségével mozgat egy `Movable` objektumot. A művelettel ezután bármely (!) olyan objektumot mozgathatunk, amely osztálya megvalósítja a `Movable` interfészt, így rendelkezik a mozgatáshoz szükséges műveletekkel.

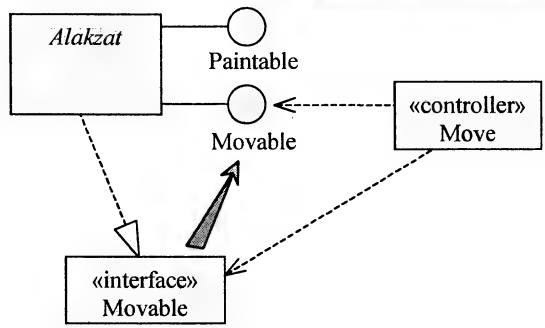
A technika fordítva is alkalmazható. Egy művelet készítésekor keressük meg a jellegzetességeknek azt a minimális körét, amelyet a működés igényel. Az objektumok alkalmazási, használati módjait, a műveletek részéről a velük szemben támasztott követelményeket fogalmazzuk meg interfészekként, így a művelet bármely olyan osztály objektumára alkalmazható, amely megvalósítja az interfészt.

jelölés

Az interfészek használatát függőséggel jelöljük. A függőség vezethet például egy osztályból vagy annak műveletéből az interfész felé. Természetesen az interfész is felhasználhat egy osztályt vagy egy másik interfészt valamely célra.

Az interfész megvalósítására az UML egy alternatív „nyalóka" *(lollipop)* jelöléssel is rendelkezik. Ez az osztály és az ikonként (körként) jelölt interfész egyszerű összekötését jelöli.

111. Ábra: Interfész és realizáció

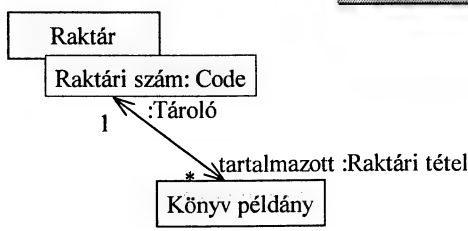


Az alternatív jelölésű interfészhez is vezethető függőség. Így nem csak az ábrázolható, hogy egy osztály megvalósít egy interfészt, illetve egy másik osztály ugyanazt az interfészt is használja, hanem azt is, hogy egy osztály egy másikat egy adott interfészen keresztül ér el.

interfész a szerepnél

Interfészek az asszociációk végén, a szerepeknél is megadhatók. Az UML szerzőinek hasonlatát követve ekkor a szerep objektumai csak bizonyos képüket mutatják az asszociáció ellentétes oldalán lévő objektumok felé.

112. Ábra: Interfész és szerep



A ***Q*** modellező módszer ajánlása szerint a szerep objektumainak típusaként minden olyan osztályt megadhatunk, amellyel konform a szerep osztálya, így annak ősosztályait is. Ha megadjuk a tároló osztály jellegét, akkor a szerep objektumainak

típusát a kisebb-nagyobb jelek közé írjuk (pl. `Set<Raktári tétel>`, kiolvasva *„set of"* `Raktári tétel`, azaz a raktári tételek halmaza).

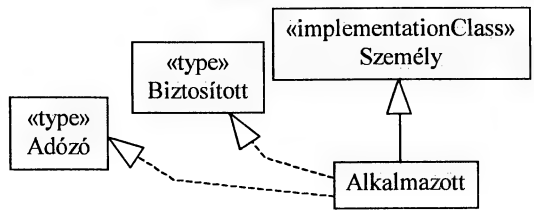
A módszer az absztrakt osztályokat és az interfészeket általánosságban felelősségekként tekinti, a lehetséges változatokkal szemben támasztott magasabb szintű követelményekként, amelyekkel felsorolhatjuk a megvalósításhoz szükséges jellegzetességeket.

dinamikus szerep

Az interfészekhez hasonló az UML «type» sztereotípiával jelölt osztálya. A «type» osztály olyan szerepet jelent, amelyet az osztály felvehet, illetve amelyről később lemondhat. Ellentéte az «implementationClass» sztereotípiával jelölt implementációs osztály, amely nem változhat.

Egy osztály legfeljebb egy implementációs osztály, de tetszőleges sok szerepjellegű osztály származtatása lehet. A «type» osztályból a származtatást realizációval jelöljük, amely ebben az esetben azt jelenti, hogy az alosztály csak a műveleteket veszi át, de az attribútumokat és az asszociációkat nem, így azokat meg kell ismételni az alosztályban is.

113. Ábra: Típus és implementációs osztály



A «type» osztályok az alosztály különböző lehetséges szerepeit jelölik, melyek eltérő jellegzetességekkel rendelkezhetnek.

Generikus osztály

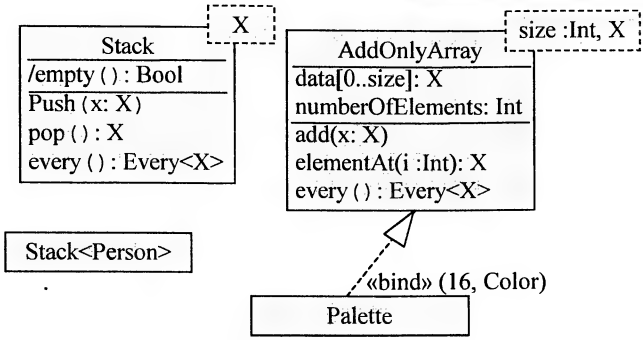
generikus szerkezet

A *generikus (generic) szerkezetekkel*, más néven *mintákkal (template)* olyan általános definíciókat fogalmazhatunk meg, amelyeket értékekkel vagy szövegekkel, például osztályok, elemi típusok vagy műveletek elnevezésével paraméterezünk.

A *generikus osztályokat* minta *(template class)* vagy paraméterezett osztályoknak *(parameterized class)* is nevezzük. A technika különösen a kollekciók megvalósításával kapcsolatos *tároló osztályok (container class)* és *felsorolások* (iterátorok) megadásánál hasznos.

Az UML a generikus osztály paramétereit a téglalap balfelső sarkába elhelyezett szaggatott vonalú téglalapban vesszővel elválasztva vagy egymás alatt sorolja fel. A paraméter elnevezését követő kettőspont után a behelyettesíthető érték típusa is megadható.

114. Ábra: Generikus osztály



Több nyelv (például a C++) is ismeri a generikus osztályok jelölését. Ha a nyelvből (például a Java-ból) hiányzik ez a technika, akkor a jelöléssel a használati módot ábrázolhatjuk (például: `Vector<Location>`, `TextField<Double>`, `Hashtable<key :String, value :Client>`).

Különleges műveletek

Az objektum létrehozását konstrukciónak, lebontását pedig destrukciónak nevezzük. Az objektum információtartalma alapján a konverzió segítségével más objektumot hozhatunk létre. Ugyancsak konverziónak nevezzük, ha az objektumot egy általánosabb vagy pontosabb változatként jelenítjük meg. A konverzióhoz hasonló módon olyan műveleteket is definiálhatunk, melyek azonos szignatúrájúak, de a visszatérési értékük különböző típusú.

Az alkalmazás működtetéséhez további, az objektumok struktúrájával és kezelésének technikájával kapcsolatos műveletek is szükségesek, melyek például új objektumok készítését, azok törlését, másolását vagy konverzióját teszik lehetővé.

Értékek másolása és ellenőrzése

érték-objektumok másolása

Értéket tároló («value») osztályok esetén megengedett az értékek másolása. Mivel egy érték több helyen is szerepelhet, ezért általában annak ellenőrzése is szükséges, hogy két érték megegyezik-e vagy sem. Az egyenlőség vizsgálata különbözik az egyediséggel rendelkező objektumok azonosságának ellenőrzésétől, amit az OO környezet biztosít.

operátorok

A ***Q*** módszer megengedi, hogy a műveletek között az osztályra vonatkozó operátorokat is definiáljunk. Az operátorok karaktereit idézőjelek között adjuk meg. Ha a nyelv nem ismeri az operátorok túlterhelésének *(operator overloading)* lehetőségét, akkor a jelekhez azonosítókat rendelünk, például:

"`:=`"	`set`,
"`=`"	`eq` vagy `equals` vagy `equals to`,
"`!=`"	`ne` vagy `not equals`,
"`<=`"	`le` vagy `less or equal`,
"`+`"	`add`,
"`+=`"	`Add`, stb.

Érték osztály (például egy szélességet és hosszúságot tartalmazó `Dimension`) esetén a ***Q*** módszer a következő alapműveleteket definiálja:

```
+" :=" (dimension: Dimension)
+" =" (dimension: Dimension): Bool
+" <=" (dimension: Dimension): Bool
```

Valamint a

" `!=`", vagy `ne` vagy `not equals`,
" `<`", vagy `lt` vagy `less than`,
" `>`", vagy `gt` vagy `greater than`,
" `>=`", vagy `ge` vagy `greater or equal` operátorokat,

melyek az " `=`" és " `<=`" relációs operátorokból már számíthatók.

A `":="` vagy `set` átírja az értéket az objektumba, az `"="` vagy `eq` vagy `equals` nevű ellenőrzi két érték egyenlőségét, a `"<="` vagy `le` vagy `less or equal` nevű ellenőrzi, hogy az érték nagyobb vagy egyenlő-e mint egy másik. Az egyenlő és a nagyobb vagy egyenlő relációs műveletek vizsgálata alapértelmezésben a két objektum attribútumértékeinek páronkénti vizsgálatával történik. A többi relációs művelet alapértelmezésben ezt a két alapműveletet használja fel.

Konstrukció és destrukció

konstruktor és destruktor

Adott osztály objektumának létrehozását *konstrukciónak* *(construction)*, megszüntetését, lebontását pedig *destrukciónak* *(destruction)* nevezzük. Az OO környezetekben az osztályoknak általában megadható a konstrukciós művelete, a *konstruktora* *(constructor)*, mely legtöbbször az attribútumait és kapcsolatait állítja be egy kezdeti értékre. A destrukciós művelettel, a *destruktorral* *(destructor)* a törléskor végrehajtandó tevékenységeket határozhatjuk meg, például egy kompozit esetén a komponensek törlését.

A konstruktornak nem adhatunk meg visszatérési értéket, mivel úgy tekintjük, hogy az maga az újonnan elkészített objektum. A destruktornak sincs visszatérési értéke, mivel az mindössze az objektum lebontásakor végrehajtandó tevékenységeket határozza meg.

jelölés

Az UML nem tartalmaz külön jelölést a konstruktorra és a destruktorra, így az alkalmazott nyelv szintaktikáját kell használnunk. A C++ és a Java nyelvben a *konstruktort* az osztály elnevezésével azonos nevű műveletként kell megadnunk, melynek nincs visszatérési értéke, például egy `X` osztály esetén `X()`. A C++ a *destruktort* a „~" szimbólumból és az osztály nevéből képzi `~X()` módon, melynek nem lehet paramétere, sem visszatérési értéke. A Java nyelvben az objektum lebontásakor az osztály `finalize` művelete hívódik meg, de ez a lebontás a „szemétgyűjtögetési" *(garbage collection)* eljárás miatt az objektum használatának befejezése után csak valamivel később következik be. Ha az objektum használatának befejeződésekor azonnal végre szeretnénk hajtani bizonyos műveleteket, akkor azt egy külön függvényben kell megadnunk, melyet közvetlenül, explicit módon hívunk meg.

A *Q* modellező módszer a konstruktorra és destruktorra külön jelöléssel rendelkezik, így azok egyszerűen azonosíthatók és az osztály nevének megváltoztatásakor ezeknek a műveleteknek az elnevezését nem kell módosítanunk.

A konstruktort a ◂ szimbólummal (szövegesen: <) vagy az `Init` nevű művelettel, a destruktort pedig a ▸ szimbólummal (szövegesen: >) vagy a `Done` nevű, paraméternélküli művelettel jelöljük; egyiknek sincs visszatérési értéke. Például:

```
◂ ()                          // alapértelmezett konstruktor
◂ (location :Location)        // másoló konstruktor
◂ (x :Int, y :Int =0)         // értékosztály kibontása
▸                             // destruktor
```

Egy osztálynak tetszőleges számú, különböző paraméterű konstruktora lehet, de legfeljebb egy destruktora. Két konstruktort különlegesnek tekintünk:

alapértelmezett konstruktor

- A paraméternélküli konstruktort *alapértelmezett konstruktornak* (`default constructor`) nevezzük, mivel az objektum attribútumait és kapcsolatait egy alapértelmezett értékre állítja.

másoló konstruktor

- Az osztállyal azonos paraméterű konstruktor a *másoló konstruktor* (`copy constructor`), mivel azonos típusú objektumból állít elő egy másik objektumot.

Értékosztályok («value») esetén létezőnek tekintjük a másoló konstruktort.

Értékosztály esetén minden konstruktornak megfelel egy azonos paraméterekkel rendelkező `set` művelet, valamint egyetlen paraméter esetén a "`:=`" operátor, amely nem az érték létrehozását, hanem az átmásolását teszi lehetővé.

Kezdőértékek értelmezése

konstruktor segítségével

A ***Q*** módszer a konstruktorok segítségével értelmezi a kezdőértékeket. Egy osztály attribútumának `a: A = v` megadása esetén úgy tekintjük, hogy új objektum készítésekor az attribútumot az `A(v)` konstruktor inicializálja. Az UML jelöléseivel több értéket zárójelek között, vesszővel elválasztva sorolhatunk fel, pl.: `p :Point =(3,4)`. Ezt a változatot is a megfelelő, többparaméterű konstruktorként értelmezzük: `Point(3,4)`. A konstrukciót csak akkor kell kiírnunk, ha kezdőértékként valamely alosztály objektumát szeretnénk értékül adni, pl.: `person :Person = Female()`.

aggregációs vagy asszociációs kapcsolat

Aggregációs vagy asszociációs kapcsolattal jelölt attribútum esetén az inicializálás során felvett objektum a dinamikus tárterületen helyezkedik el, ezért a kezdőértékadást egy új, részben független objektum létrehozásaként kell értelmezni (például: `new Point(3,4)`).

1 számosságú attribútum

Ha nem adunk meg kezdőértéket, akkor 1 számosság esetén úgy tekintjük, hogy a típusnak megfelelő alapértelmezett konstruktor hajtódik végre. Elemi értékek „alapértelmezett" konstruktoraként az alapértéket tekintjük, ami számok esetén 0, logikai érték esetén hamis, stb., ezért ezeket az inicializálásokat felesleges kiírni.

összetett kezdőérték

Több érték összekapcsolásának jelölésére az UML ugyancsak a zárójelet alkalmazza, pl.: `p :Path = ((0,0),(5,0),(5,5),(0,5),(0,0))`. A ***Q*** módszer ekkor a kapcsos-zárójelet javasolja, melyet iterátorként vagy kollekcióként (mely lehet egyszerű tömb is) értelmezünk, a konstruktor paraméterének megfelelően (pl.: a `Path` osztály ◂ `(List<Point>)` vagy ◂ `(Every<Point>)` konstruktoraként. A C nyelv a kapcsoszárójelet ugyancsak az összetett kezdetiérték megadására, a tömb inicializálására használja.

A módszer a kezdőértéket *alapértelmezett értékként* is tekinti, így nem történik tényleges inicializálás, ha valamely konstruktor egy attól eltérő kezdőértéket határoz meg.

A módszer egy érdekessége, hogy a konstrukcióval azonos módon a visszatérési értékkel rendelkező műveleteknek is jelölhető az alapértelmezett/kezdeti értéke, pl.: `fn() :Int(5)`, vagy `y() :Point(3,4)`. *visszatérési érték*

A technika a gyakorlatba úgy ültethető át, hogy felveszünk egy adott nevű, a visszatérési típusnak megfelelő változót, amit a művelet elején a megfelelő értékkel inicializálunk. Visszatéréskor a művelet a változónak megfelelő értéket utal át.

Egyszerűbb esetekben a művelet a „kezdőértékként" megadott egyetlen kifejezésre redukálható: pl.: `kerület() :Real(2*(a+b))`.

Konverzió

A ***Q*** módszer alapvetően kétfajta *átalakítást*, *konverziót* (`conversion`) határoz meg. *Értékjellegű* konverzió esetén egy objektum információtartalma alapján egy másik értéket vagy objektumot készítünk. Az értékjellegű konverzió az iránya alapján lehet az *osztályba* és az *osztályból* történő konverzió. *értékjellegű konverzió*

- Az *„osztályba"*, azaz az osztály objektumává történő konverzió során bizonyos értékekből az osztály objektumát állítjuk elő, ez pedig nem más, mint a konstruktor. Az osztályba történő konverzió ezért a megfelelő paraméterű konstruktorral adható meg.
- Az *„osztályból"* történő konverzió során az osztály objektumának információtartalmából egy másik értéket vagy objektumot állítunk elő.

Az osztályból történő konverziót a ***Q*** modellező módszer a ▸ szimbólummal jelöli. A konverzió eredményének típusát a visszatérési érték típusaként adjuk meg. A konverzió a kódban egyrészt konverziós operátorként jelenik meg (ha a nyelv erre lehetőséget ad), másrészt, például a `▸:String` konverzió esetén a következő műveletként:

```
toString() :String
```

A módszer az alapértelmezett konstruktort a „semmiből" történő konverziónak, konstrukciónak tekinti, a destruktort pedig a „semmibe" történő átalakításnak. *konstruktor és destruktor*

Az átalakítás másik fajtája, a *típuskonverzió*. Ekkor ugyanazt az objektumot egy ősosztály vagy valamely érvényes alváltozat objektumaként tekintjük. A típuskonverziót az „as" prefix és a típusnév segítségével képezzük. Például egy `Személy` osztály objektumát annak `asOrvos() :Orvos` műveletével érhetjük el `Orvos` alosztályként, ha az valóban az alosztálynak megfelelő változatú. *típuskonverzió*

A műveletek szignatúrájába az OO környezetek, sőt az UML sem veszi be a visszatérési érték típusát, a ***Q*** modellező módszer azonban igen. Egy adott műveletet követően az azonos szignatúrájú, de eltérő típusú műveletet az `mAsT` módon nevezhetünk át, ahol `m` az eredeti művelet, `T` pedig a visszatérési érték típusa, például `getReportAsText() :Text`. *visszatérési érték típusa*

Kiegészítő technikák

Az alapvető jelölésekkel jól ábrázolható az alkalmazás belső felépítésének vázlata. A modellt néhány kiegészítő jelöléssel tovább finomíthatjuk. Osztályattribútumokként összetett konstansokat vehetünk fel, osztályműveletekkel pedig az osztály konkrét objektumától független megoldásokat, módszereket adhatunk meg. Invariánsokkal, valamint elő- és utófeltételekkel a helyes működés esetén a modellelemekre vonatkozó szabályokat rögzítünk. A műveletek végrehajtási módját megszorításként jelölhetjük.

Osztályattribútumok és osztályműveletek

osztály-attribútum

Osztályattribútumok `(class attribute)` segítségével az osztály minden objektumára érvényes értékeket tárolhatunk. Ezek hasonlóak a hagyományos programozási nyelvek globális változóihoz, de az osztály el is rejtheti ezeket az attribútumokat védettként `(protected)` vagy sajátként `(private)` megjelölve.

Egy osztályattribútum alkalmas például az osztály objektumainak számontartására vagy egyszerűen a létező példányok számának tárolására.

osztály-művelet

Az *osztályművelet* `(class method)` az osztályra vonatkozó műveletet jelent, ezért közvetlenül kizárólag az osztályattribútumokat, illetve a többi osztályműveletet éri el. Egy osztályművelet csak közvetve hivatkozhat a többi, un. *példányattribútumra* `(instance attribute)` és *példányműveletre* `(instance operation)`, például ha újonnan készít, paraméterként megkap, vagy az osztályattribútumon keresztül elér egy objektumot. Az osztályműveleteket a hagyományos programozási nyelvek (globális) eljárásaihoz és függvényeihez hasonlíthatjuk, de az osztály a láthatóság korlátozásával a külvilág elől el is rejtheti ezeket a műveleteket.

példány-képzés

Az osztály új objektumát létrehozó konstrukciós művelet, más néven a példányképzés `(instantiation)` valójában egy osztályművelet, de ezt külön nem jelöljük.

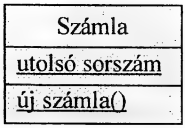
Az UML az osztályattribútumokat és az osztálymőveleteket a definíció aláhúzásával jelöli. Jegyezzük meg, hogy az OMT az elnevezést megelőző `$` jelet alkalmazta.

Az osztályattribútumok és osztályműveletek nem csak egy adott objektum esetén érhetők el (például `l.balra`, ahol az `l` egy `Location`), hanem azokra az osztályon keresztül is hivatkozhatunk (pl.: `l.compare(l2)==Location.balra`).

Gyakori szervezési hiba az osztályattribútumok és osztályműveletek helytelen használata. Egy `Számla` osztály esetén az utolsó számla sorszámát felvehetjük osztályattribútumként, mivel az az osztály minden objektumára érvényes érték.

Hasonlóan, az utolsó sorszám segítségével történő új számla felvitele sem egy konkrét `Számlára` vonatkozó művelet, ezért azt osztályműveletként jelölhetjük.

115. Ábra: Osztályattribútum és osztálymővelet



modell ellenőrzése

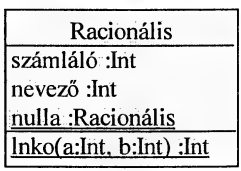
Ha jobban megvizsgáljuk ezt a két jellegzetességet, akkor észrevehetjük, hogy ezek nem az osztály összes objektumára, hanem azok egy kollekciójára vonatkoznak. A modell hibája azonnal nyilvánvalóvá válik, ha nem egyetlen, hanem több céghez tartozó számlákat is tárolni szeretnénk. Ezeket a jellegzetességeket nem célszerű az osztályhoz rendelni, hanem a külön osztályként (például `Számlák`) felvett kollekcióban megadni. A későbbi tervezés során dönthetünk úgy, hogy ha az alkalmazásban a számlák csak egyetlen (esetünkben `Cég`) objektumhoz tartozhatnak, akkor a kollekció attribútumait és műveleteit osztályjellegzetességekként „beviszszük" az osztályba. Ez azonban egy nagyon kockázatos megoldás, mely igaz, hogy lerövidíti a fejlesztés idejét, de megnehezíti az alkalmazás karbantartását és későbbi továbbfejlesztését.

konstansok és módszerek

A gyakorlati tapasztalatok alapján nem követünk el szervezési hibát, ha az osztályattribútumokat összetett alapértelmezett értékek és konstansok tárolására használjuk, az osztálymőveletekkel pedig logikailag az osztályhoz tartozó, de nem a konkrét objektumra vonatkozó megoldási módokat, módszereket, „trükköket" adunk meg.

Például a `számláló/nevező` alakú racionális számok esetén a nulla konstanst megadhatjuk osztályattribútumként, amely így `Racionális.nulla` módon érhető el. Bizonyos műveletek után célszerű a racionális számot egyszerűsíteni, mely a számláló és nevező legnagyobb közös osztójának meghatározásával történik. A legnagyobb közös osztó képzése egy számítási módszer, egy „trükk", melynek nincs szoros kapcsolata az aktuális racionális számmal, ezért azt célszerű külön osztálymőveletként megadni.

116. Ábra: Összetett konstans és számítási mód



eszköztár

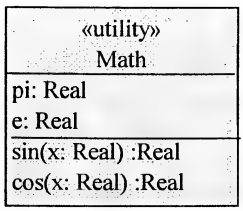
Az OO fejlesztés során a feladatok adott körével kapcsolatos, több osztályt érintő konstansokat és számítási módszereket nem globális változóként, eljárásként vagy függvényként adjuk meg, hanem azokat *eszköztár* `(utility)` osztályokba foglaljuk. Az eszköztár egy olyan speciális osztály, melynek minden jellegzetessége

osztályjellegzetesség (azaz egy konstans és számítási mód („trükk") gyűjtemény), és amelyből a példányképzést általában nem engedélyezzük.

«utility»

Az UML az eszköztár osztályok jelölésére a «utility» sztereotípiát alkalmazza, melyek minden attribútumát és műveletét osztályjellegzetességnek tekinti. Ez nem egy alapvető konstrukció, de érthetőbbé teszi a modellt.

117. Ábra: Eszköztár



objektum definíciója

Az OO nyelvek általában nem támogatják egyetlen objektum definícióját. A ***Q*** modellező módszer egy közvetett megoldási módnak tekinti az eszköztárat, illetve azt az osztályt, amelynél kikötjük, hogy csak egyetlen példánya lehet. Ezeknek a speciális „osztályoknak" a segítségével az alkalmazás szerkezeti leírásába valójában objektumokat csempészhetünk be.

Hasonlóan objektumok az alkalmazás futtatható részeiként, programjaiként megjelenő csomagok és egyes felhasználói felületek is.

A ***Q*** módszer javaslata szerint az alkalmazás egészére vonatkozó objektumokat a strukturális leírásba egy általános «instance», illetve futtatható állományok esetén «application» sztereotípia segítségével vigyük be az eszköztárak «utility» jelölése mellett. Ezekben az esetekben minden jellegzetességet osztályjellegzetességként tekintünk és nem engedélyezzük az osztály pontosítását és objektumának készítését. Itt is megadható konstruktor és destruktor, ezek azonban az alkalmazás indításakor és leállításakor hajtódnak végre, így azokat a programozási nyelvek esetén más módon kell megvalósítanunk (Java nyelven a konstruktort például statikus inicializálásként).

Metaosztály

osztályra vonatkozó információ

Metaosztálynak (metaclass) nevezzük azt az osztályt, amelynek a példányai osztályok. Egyes objektumorientált környezetekben és programozási nyelvekben (például a Java és a Smalltalk nyelvben) az osztályok szerkezetét, például attribútumait és műveleteit működés közben is el lehet érni. Az osztályokra vonatkozó információt a metaosztályok (szó szerinti fordításban: „az osztályokon túli osztályok") objektumai tárolják.

«metaclass»

A metaosztályokat az UML a «metaclass» sztereotípiával jelöli.

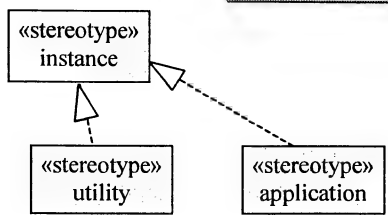
Az osztályt és a szerkezetét leíró metaosztályt az «instanceOf» sztereotípiájú függőséggel kapcsolhatjuk össze.

Sztereotípia

A sztereotípiák között is meghatározhatunk általánosítás/pontosítás viszonyokat. Az UML jelöléseivel ekkor a sztereotípiát osztályként ábrázoljuk, melyet a «stereotype», azaz „sztereotípia" sztereotípiával jelölünk.

«stereotype»

118. Ábra: Sztereotípiák ábrázolása



Invariánsok, elő- és utófeltételek

A modellező nyelvek jelöléseivel a modellünkre vonatkozó információkat fogalmazhatjuk meg. Ha egy ábrázolni kívánt részletre nincs külön jelölésünk, akkor azt a modellbe *megszorításként* (constraint) vehetjük fel. A megszorító szabályt az UML-ben kapcsoszárójelek között adhatjuk meg, mely lehet képlet, egy programozási nyelv kifejezése vagy egyszerű szöveg.

Az osztályokat, illetve azok attribútumait és műveleteit egy magasabb absztrakciós szinten jellemezhetjük három speciális megszorítás, az invariánsok, az elő- és az utófeltételek segítségével. A technikát (design by contract) legrészletesebben Bertrand Meyer publikálta.

Az *invariánsok* (invariant) strukturális elemek (osztályok, attribútumok, asszociációk, stb.) esetén adhatók meg. A szabálynak minden konkrét esetben (például egy osztály minden objektuma esetén) teljesülni kell, ha az elem elérhető. Egy osztály objektumaira vonatkozó invariáns sérülhet ugyan egy műveleten belül, de azt helyre kell állítani, mielőtt az objektum más üzeneteket is fogadna.

invariáns

Az *elő-* és *utófeltételeket* (pre- and postcondition) műveletek esetén adhatjuk meg. Mindkettő a művelet környezetére vonatkozó szabály. Az előfeltétel leírja, hogy a művelet milyen környezetben (például milyen paraméterértékek esetén) értelmezhető, az utófeltétel pedig az előfeltételnek megfelelő környezet esetén meghatározza a művelet végrehajtása után a környezetre és a visszaadott értékekre teljesülő szabályt. Esetenként a művelet annyira egyszerű átalakítást vagy lekérdezést jelent, hogy az utófeltétel magát a műveletet is egzaktul meghatározza.

elő- és utófeltétel

A pontosítás során az invariánsokat és az utófeltételeket legfeljebb tovább korlátozhatjuk („erősíthetjük"), míg az előfeltételeket legfeljebb gyengíthetjük. Egy osztály példányaira vonatkozó invariánsnak az osztály műveletének meghívása előtt és után is teljesülnie kell, így azt nem szükséges befoglalni az elő- és utófeltételekbe.

pontosítás

Az UML ajánlása szerint a szabályokat külön «invariant», «precondition» vagy «postcondition» sztereotípiájú megjegyzéslapon adjuk meg, melyet az elemhez vagy az elemekhez kapcsolunk. Fowler és Scott javaslatát követve a megjegyzéslapon a szabályokat célszerű megszorításként jelölni.

hibakeresés

A legtöbb invariáns, illetve elő- és utófeltétel a program helyes működése esetén teljesül, így sérülésük a megvalósítás hibájára utal. Ezeket az állításokat általában csak hibakeresés *(debug)* során ellenőrzzük. Például egy rendezett kollekció esetén az elemek felvitele után is fent kell maradni a rendezettségnek.

Az invariánsok, illetve elő- és utófeltételek csak egyetlen nyelvbe, a Meyer által kifejlesztett Eiffel-be épültek be szervesen. Az Eiffel a szabályokat a hibakeresés bekapcsolásakor ellenőrzi, normál végrehajtás során azonban figyelmen kívül hagyja.

Ha nem egy zárt alkalmazást, hanem egy nyitott, más által is használható rendszert, például egy keretrendszert készítünk, akkor célszerű az előfeltételeket mindig ellenőrizni. A művelet nem megfelelő környezetéről és paramétereiről kivétellel tájékoztathatjuk a hívót.

A ***Q*** modellező módszer az elő- és utófeltételekre egy `pre:` illetve `post:` rövidítésű címkével jelölt megszorítást ajánl, melyben zárójelek között adjuk meg a feltétel ellentétét, melyet ^ után a hiba által kiváltott kivétel követhet, például

```
pop() :X {pre: (empty) ^EmptyStack(this)}
```

A példát a következő módon írhatjuk át Java nyelvű programrészletté (az `EmptyStack` egy `RunTimeException`, ezért a műveletnél nem kell definiálni):

```
X pop() {
    if( isEmpty() ) throw new EmptyStack(this);
    ...
```

Műveletek végrehajtási módja

A művelet végrehajtásának típusát megszorításként adhatjuk meg. Az UML a következő jellemzőket definiálja:

- a `{sequential}` *(„szekvenciális")* megszorítás egyszerű szekvenciális végrehajtást határoz meg, melyet általában alapértelmezettnek tekintünk. Szekvenciális esetben a használóknak kell biztosítaniuk, hogy egy objektumnak azonos időben (konkurensen) nem hívjuk meg több műveletét.
- `{guarded}` *(„őrzött", „felügyelt")* megszorítással jelöljük, ha a művelet feldolgozását valamely mechanizmus szabályozza, mely például egy várakozási sorba állítja a kéréseket. Az üzenet átadása ekkor az igényt mindössze a várakozási sorba veszi fel, melyből az objektum egymás után kiemeli és feldolgozza a soron következő elemet.

- a {concurrent} *(„konkurens")* megszorítással jelöljük a párhuzamosan is végrehajtható, szinkronizálást (monitort) igénylő művelet. Ebben az esetben az OO környezet biztosítja, hogy a művelet a végrehajtás idejére zárolja *(lock)* az objektumot, így a más helyről, párhuzamosan történő hívás megvárja a művelet befejezését és a zár feloldását *(unlock)*. A zárolás párhuzamos végrehajtás esetén is biztosítja az objektum integritását. A „konkurens" művelet Java nyelven a synchronized kulcsszóval jelölhető, mely a művelet végrehajtásának idejére zárolja az objektumot.

A «signal» sztereotípiával kiegészített művelet jelöli, hogy az osztály elfogadja az adott szignált. A szignálok átvétele {guarded} mechanizmussal is történhet. A szignál a művelettel azonos módon adható meg.

Az objektumorientált szemléletben általában két csoportba sorolják a műveleteket. Az atomi, azaz félbeszakíthatatlan műveletek az *akciók (action)*, melyeket úgy tekintünk, hogy egyetlen időpillanat alatt zajlanak le. Ezzel szemben a *tevékenységeket (activity)* úgy tekintjük, hogy a végrehajtásuk hosszabb időt, egy meghatározott időtartamot vesz igénybe.

akció és művelet

A *Q* módszer ajánlása szerint, ha egy művelet az osztály megvalósításának részleteire hivatkozik, akkor azt legfeljebb egy-két programsorban fogalmazzuk meg, optimális esetben egyetlen kifejezésként, értékadásként, egyszerű feltételes elágazásként vagy ciklusként definiáljuk. Az egyszerű műveleteket egyben akcióknak is tekintjük. A több lépést igénylő hosszabb műveleteket a többi műveletet felhasználva, az „absztrakt modell szintjén" adjuk meg.

egyszerű művelet

Egyszerű műveletként célszerű megfogalmazni az objektum változásának elemi transzformációit, amely az objektumot annak egy állapotából egy másik állapotába billenti át, valamint a lekérdezéseit is. A módosítás művelete időlegesen megsértheti az objektum belső integritását, invariánsait, de a befejeződés előtt azokat vissza kell állítania. A transzformációkat akcióknak, azaz félbeszakíthatatlanoknak, atomiaknak tekintjük, melyek az objektum változásának elemi lépéseit jelölik, csak azokat az adott programozási nyelven több lépésben kényszerülünk megfogalmazni.

Ha a hosszabb művelet egy tevékenység, így a végrehajtása időt vesz igénybe, illetve az félbeszakítható, akkor a műveletet lássuk el az {activity} megszorítással.

A *Q* módszer osztályok esetén is megengedi a {concurent} megszorítás használatát. Ekkor minden műveletet akciónak tekintünk, amely nincs ellátva az {activity} megszorítással. Alapértelmezésként az akciók konkurensek, azaz szinkronizáltak az „egyszer írható, többször olvasható" protokoll alapján. Ez azt jelenti, hogy az objektumon egyszerre több lekérdezés is végrehajtható, azonban egy transzformáció (módosító) kizárólagosan blokkolja az objektumot, így másik módosító vagy lekérdező műveletet sem engedélyez, illetve maga a transzformáció sem kezdődhet el, mielőtt a lekérdező műveletek befejeződnének.

szinkronizáció

Az {active} megszorítással jelölt önálló, párhuzamos működésű aktív osztályokat ugyancsak konkurenseknek tekintjük.

Objektumok

Objektumok segítségével szemléltethetünk egyes helyzeteket, illetve az alkalmazás adott időpillanatban felvett állapotának bizonyos részleteit. Jelölhetjük az objektumok attribútumértékeit, állapotukat, a közöttük fennálló kapcsolatokat, valamint a kompozit objektumok belső felépítését. A diagramon az objektum időbeli változását a «become» függőséggel ábrázoljuk.

objektumdiagram

konkrét vagy jellemző helyzet

Egy adott időpillanat konkrét objektumait és a közöttük lévő kapcsolatokat az UML *objektumdiagramja* (object diagram) segítségével ábrázolhatjuk. Objektumokkal nem csak egy konkrét szituációt, hanem egy *jellemző helyzetet* is felvázolhatunk. Ebben az esetben az objektumok nem konkrét elemeket, hanem szerepeket jelölnek, a diagram által meghatározott minta olyan „réseit" (slot), változóit, amelyekbe konkrét objektumok helyettesíthetők.

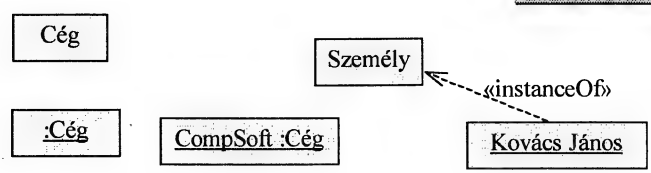
Az UML az objektumok ábrázolására egy külön diagramtípust definiál. Mi a következőkben objektumdiagramnak egyszerűen egy olyan osztálydiagramot tekintünk, amelyen szerepel legalább egy objektum.

Objektumok

jelölés

Az UML jelöléseivel egy objektumnak megadhatjuk az azonosító nevét, egy kettőspontot követően az osztályát vagy mindkettőt. A példány jelleget az azonosító név, illetve az osztály aláhúzásával jelöljük.

119. Ábra: Objektum



«instanceOf»

Az objektum osztálya egy «instanceOf» sztereotípiájú függőséggel is jelölhető, amely az objektumból az osztálya felé vezet (mivel az objektumnak ismernie kell az osztályát, azaz használja az abban rögzített definíciókat). Ez a forma egyenértékű a kettőspont után írt osztálynévvel.

objektum neve

Az objektum neve nem egy (például név) attribútumának az értékét jelenti, hanem valami olyasmit, amellyel mi, a diagram olvasói egyértelműen azonosítjuk az objektumot, tehát a szerepe az OO környezet objektumazonosítójához hasonló. Névtelen, anonymous objektum esetén (például :Cég, kiolvasva: „egy Cég") a vázlatunkban lényegtelen az azonosító név.

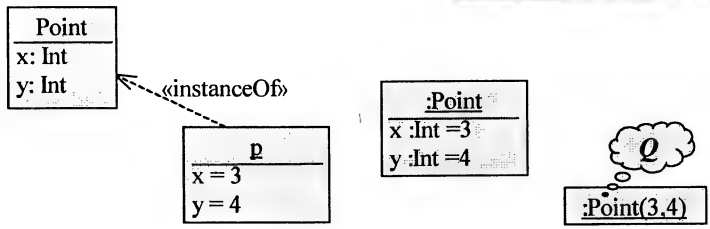
Objektum attribútumértékei és állapota

attribútum-értékek

Az ábrázolt objektumokat néhány kiválasztott attribútumuk értékének megadásával jellemezhetjük. Az értéket az attribútumnév és egy egyenlőségjel után adhatjuk meg. Nem szükséges az összes attribútumot felsorolni, elegendő csak azokat, amelyek lényegesek a helyzet jellemzésének szempontjából.

A nevet követő kettőspont után az attribútum típusa is megadható, ha azt hangsúlyozni szeretnénk. A típusnak természetesen meg kell egyezni az eredeti attribútum típusával. A jelölést akkor is felhasználhatjuk, ha egy még definiálatlan osztályú konkrét objektumot szeretnénk ábrázolni.

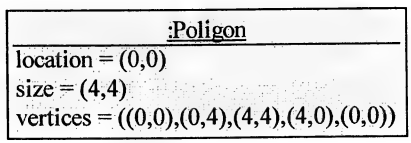
120. Ábra: Attribútumértékek



A *Q* módszer megengedi az objektumnak egy, a névrészben elhelyezett konstruktor-jellegű rövidített jelölését is, illetve a kezdőértékadással teljesen azonos szintaktikát javasol.

Az UML jelöléseivel az összetett értéket zárójelek között kell megadnunk.

121. Ábra: Összetett érték

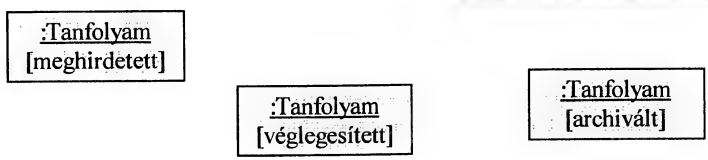


objektum állapota

A konkrét attribútumértékek meghatározása mellett az objektum az állapota segítségével is jellemezhető. Az állapot általában több lehetséges attribútumérték kombinációjaként jelenik meg.

Az állapotot az objektum elnevezése, illetve osztálya alatt, szögletes zárójelek között adhatjuk meg.

122. Ábra: Objektum állapota

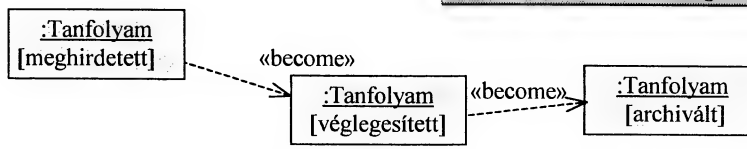


Az «instanceOf» sztereotípiájú függőség mellett az UML az objektumok esetén használható további két függőséget definiál:

«become»

- «become» függőséggel ábrázolhatjuk az objektum időbeli változását.
- «copy» függőséggel jelöljük, ha a függő elem az eredetinek egy másik helyen megjelenő másolata.

123. Ábra: Időbeliség ábrázolása

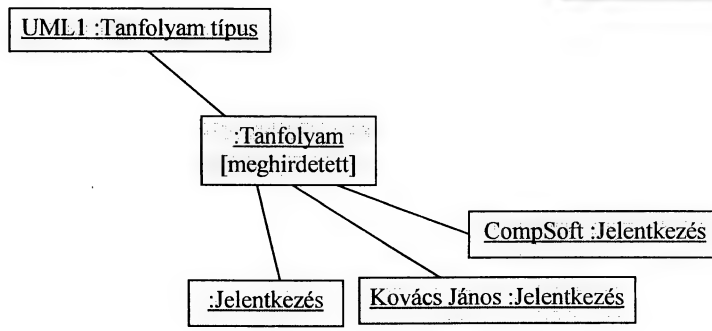


Objektumok és kapcsolatok

Az objektumok között az elérés tényleges lehetőségét, illetve az összetartozást kapcsolatokként ábrázoljuk.

Az UML az objektumok közötti kapcsolatokat az asszociációkhoz hasonlóan, vonallal jelöli.

124. Ábra: Kapcsolat



kapcsolat jellege

A kapcsolat jellegét sztereotípiaként adhatjuk meg. Az UML a következő négy kapcsolatot definiálja:

- «association» *(„asszociáció")* egy strukturális kapcsolatot, a két objektum osztálya közötti asszociáció példányát jelöli. Kapcsolatok esetén ez az alapértelmezett, így csak akkor adjuk meg, ha hangsúlyozni szeretnénk, hogy a kapcsolat nem valamely más típusú.
- «self» *(„önmaga")* egy önmagára mutató kapcsolatot jelöl. Minden objektum természetesen eléri önmagát, így például felhasználhatja bizonyos műveleteit valamely más tevékenységének megvalósításához.
- «global» *(„globális")*: a globális objektum bármely más objektumból is elérhető. Valódi objektumorientált rendszerben nincsenek globális változók, azonban ezt a szabványos jelölést felhasználhatjuk egy osztályváltozó vagy osztálymővelet elérésének jelölésére is.

- «local» *(„lokális", „helyi" változó)*: adott objektum művelete létrehozhat és így el is érhet lokális objektumokat.
- «parameter» *(„paraméter")* a művelet paramétereként megkapott objektum is elérhető a művelet végrehajtásának ideje alatt.

élettartam

Egy művelet létrehozhat, illetve megszűntethet más objektumokat. Például egy művelet végrehajtásának szemléltetésekor a kapcsolat megfelelő végén elhelyezett megszorítással jelölhetjük az élettartam szabályozását. Az UML a következő három lehetőséget definiálja:

- {new} *(„új")*: az objektum a művelet során jön létre.
- {destroyed} *(„törölt")*: a művelet során megszűnik az objektum.
- {transient} *(„tranziens")*: az objektum csak a művelet (az ábrázolt helyzet) alatt létezik, így ez a {new destroyed} rövidítése.

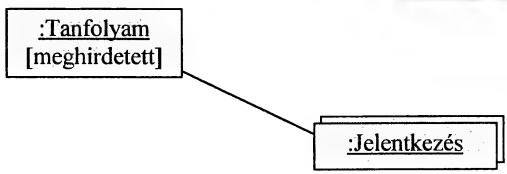
Kapcsolatok esetén is megadható név, szerepnév, aggregációs vagy kompozíciós jel és navigációs irány, azonban számosság (multiplicitás) nem jelölhető.

multi-objektum

Az UML a *multiobjektum* (multiobject) alakzatát vezeti be, ha egy objektum több, azonos típusú objektummal való kapcsolatát szeretnénk ábrázolni.

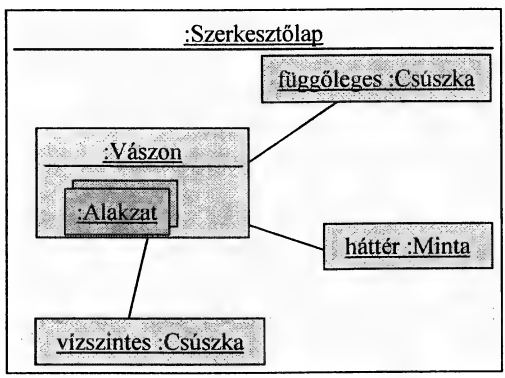
A multiobjektum korábbi szóhasználatunk alapján egy objektumkollekció, azaz egy kollekció konkrét változata.

125. Ábra: Multiobjektum



Kompozit objektum egy szemléletesebb „beágyazott" jelöléssel is ábrázolható.

126. Ábra: Kompozit objektum



passzív és aktív objektum

Egy helyzet szemléltetése során megkülönböztethetjük a passzív (várakozó) és az aktív (működő) objektumokat. A vastagított oldalú téglalappal jelölt objektum rendelkezik a vezérléssel, azaz működik, így más objektumoknak is küldhet üzenetet, míg a passzív objektum csak hívás esetén kapja meg a vezérlést.

{active}

Az aktív objektumot az `{active}` megszorítással is jelölhetjük.

127. Ábra: Aktív objektum

Osztálydiagramok használata

Osztálydiagramok segítségével ábrázoljuk az alkalmazás belső szerkezetét, az architektúrát. Az osztálydiagramok a fejlesztés különböző fázisaiban másképpen használhatók, így rögzíthetik a szakterület fogalmait és a közöttük lévő viszonyokat, az alkalmazás megvalósításához szükséges technológiai kiegészítéseket, vagy tekinthetők a forráskód áttekintő térképeiként.

alkalmazás alapszerkezete

Az osztálydiagramok alapvető szerepűek az objektumorientált fejlesztés során, mivel segítségükkel ábrázolhatjuk alkalmazásunk alapszerkezetét. A használati eset diagramok az alkalmazással szemben támasztott követelmények áttekintésére alkalmasak, tehát a felhasználók, az igények szempontjából rajzolják meg a rendszer határait. A mérnöki, rendszerszervezői oldalról közelítő osztálydiagramok esetén már a megvalósítás kerül előtérbe. Az osztálydiagramok nem az alkalmazás külső képét, hanem belső felépítését, szerkezetét, az architektúrát ábrázolják, ezért nem a felhasználók, hanem a fejlesztők szempontját képviselik.

Az osztálydiagramokon megfogalmazott információk köré csoportosítható az UML néhány további diagramtípusa, az interakciós, az aktivitás és az állapotdiagram.

diagramelemek

Az osztálydiagramokon a következő elemek szerepelhetnek:

- a csomópontokban az osztályokat helyezzük el, melyeket
  - az attribútumaikkal és
  - a műveleteikkel adunk meg,
- az elemek közötti viszonyokat élekkel ábrázoljuk, melyek lehetnek
  - asszociációk,
  - általánosítás/pontosítás viszonyok,
  - függőségek és
  - megszorítások (szövegesen megfogalmazott viszonyok).

használat

Az osztálydiagram a fejlesztés különböző fázisaiban másként használható:

- Az elemzés során a szakterület fogalmait és a közöttük lévő viszonyokat ábrázolhatjuk.
- A tervezés fázisában meghatározzuk a szakterületi modell megvalósításának módját és az azt működtető infrastruktúrát. A szakterületi modellt kiegészítjük a megvalósításhoz szükséges technológiai apparátussal.
- A megvalósításkor az implementáció bizonyos részleteit szemléltethetjük. Ekkor az osztálydiagramokat a forráskód összefoglaló, áttekintő térképeiként használjuk.

III.

Interakció-diagramok

Interakció-diagramok segítségével az objektumok nem strukturális kapcsolatait a közöttük lefolytatott kommunikáció alapján jellemezzük. A diagramokkal az összetettebb feladatok megvalósítása során együttműködő objektumokat ábrázoljuk, ahol az együttműködést üzenetváltások segítségével valósítjuk meg. Különleges üzenet az objektumok létrehozása és törlése.

UML két interakció-diagram típust definiál. Szekvencia-diagramokkal az üzenetek időbeli viszonyait emelhetjük ki. Az együttműködési diagramok az objektumok szerveződésére, kapcsolódási módjaira helyezik a hangsúlyt.

Interakció-diagramok

Az összetettebb feladatok megvalósítása során együttműködő objektumokat interakció-diagramok segítségével ábrázolhatjuk. Az UML két interakciós diagramtípust határoz meg. Szekvencia-diagramokkal a folyamat időbelisége, együttműködési diagramokkal pedig a feladatot megvalósító elemek szerkezeti felépítése jeleníthető meg.

együttműködés részletei

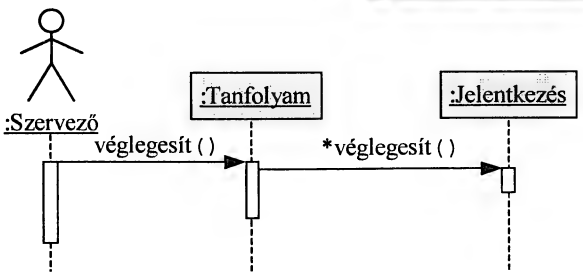
Az *interakció-diagramok* `(interaction diagram)` segítségével leírhatjuk, hogy adott objektumok adott körülmények között, például egy összetett feladat megvalósításakor hogyan működnek együtt. Egy interakció-diagram általában egy használati eset részleteit határozza meg.

A ***Q*** modellező módszer javaslata szerint az interakciókat a csomagok, illetve az osztályok használati eseteinek jelölésével is vázolhatjuk. Ennek a megközelítésnek az előnye, hogy összhangban van más jelölésekkel és az együttműködés könnyen áttekinthető és követhető. Hátránya azonban, hogy eltér az UML megszokott használatától és a szabványos jelölések keretein belül maradva csak kevés részlet ábrázolható. Például egy függőség esetén nem adhatunk meg multiplicitást, holott a ***Q*** módszer „használat" értelmezésében ez szükséges lenne.

időbeliség

Szabványosan az interakciók két diagramtípussal ábrázolhatók. *Szekvencia-diagramokkal* `(sequence diagram)` a folyamat időbeliségét emelhetjük ki.

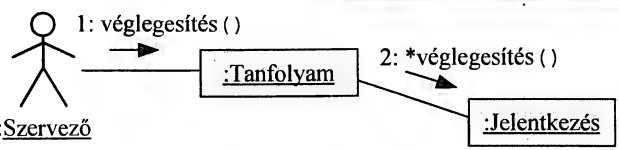
128. Ábra: Szekvencia-diagram



szerkezeti felépítés

Együttműködési diagramok `(collaboration diagram)` alkalmazásával az együttműködő elemek szerkezeti felépítését hangsúlyozzuk.

129. Ábra: Együttműködési diagram



Mindkét diagramtípuson a csomópontokban elhelyezett alapelemek a *példaobjektumok* `(example object)`. Az együttműködést megvalósító *üzeneteket* `(message)` a példaobjektumokat összekötő élekkel ábrázoljuk.

Példaobjektum

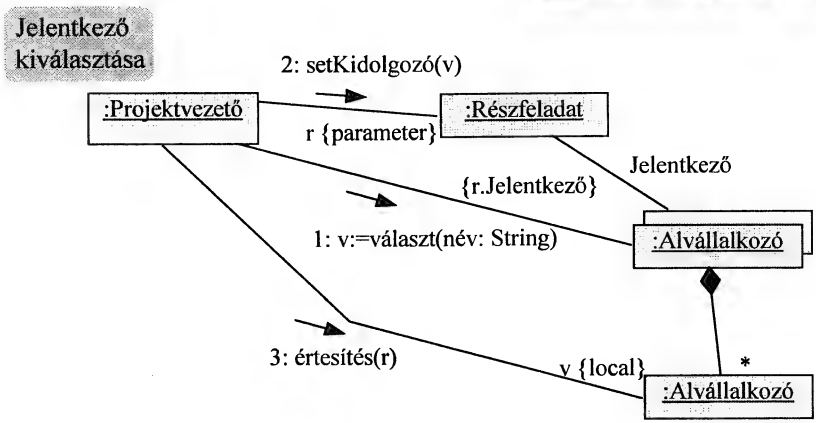
Az interakciós diagramok alapelemei a példaobjektumok. Interakcióként legtöbbször egy jellemző helyzetet, például egy művelet végrehajtása során az együttműködő objektumok közötti üzenetváltásokat ábrázoljuk. A példaobjektumok ezért általában szerepeket, „változókat" jelölnek, melyek egy-egy konkrét esetben konkrét objektumokkal helyettesítődnek.

szerepet jelöl

Ha egy üzenet több, azonos osztályhoz tartozó objektumot, azaz egy kollekciót címez meg, akkor azt *multiobjektumként* `(multiobject)` ábrázoljuk. A jelölés másik lehetősége az objektum téglalapjának jobb-felső sarkába írt több (*) számosság.

multiobjektum

130. Ábra: Multiobjektum



A vastag kerettel vagy az `{active}` megszorítással jelölt *aktív* `(active)` objektum rendelkezik a vezérléssel, és így üzenetet küldhet más objektumoknak. Az aktív objektum ellentéte a *passzív objektum*, amelynek alapállapotában felfüggesztett a működése és csak egy üzenet hatására, kívülről aktiválódik és kezdi el annak feldolgozását.

aktív objektum

Üzenet

Az objektumok az együttműködést üzenetváltásokkal, *üzenetek* `(message)` küldésével és fogadásával, más néven interakciókkal valósítják meg. Az *üzenetküldést* `(message passing)` a két diagramtípus eltérő módon ábrázolja, az üzenet megadása azonban azonos módon történik.

üzenet és üzenetküldés

Mindkét esetben az üzenet irányát a küldőtől a fogadó felé mutató nyíl jelöli.

jelölés

Egy üzenetküldést a következő alapelemekkel adhatunk meg:

- *vezérlő információk.*
- *visszatérési érték* vagy értékek `v1,v2:=` formában, amely elnevezésekkel a későbbiekben az üzenet eredményeire hivatkozhatunk.
- az üzenet *elnevezése* (ezt minden üzenet esetén célszerű megadni),
- az üzenet *argumentumai* (más néven paraméterei) zárójelek között.

Az üzenet küldése tehát egy művelet hívásának a szintaktikáját követi, de azt nem feltétlenül metódushívással valósítjuk meg.

vezérlés

A vezérlőinformációk az üzenet elküldésének számát határozzák meg. A két legfontosabb vezérlés:

- a *feltétel* `(condition)`, amely az üzenet küldésének feltételét adja meg. Jele az üzenet neve elé, szögletes-zárójelek közé írt logikai kifejezés, pl.: `[x>y]`. Alternatív leágazás esetén vagy a feltétel ellentétét fogalmazzuk meg, vagy az `else` kulcsszót alkalmazzuk.
- Az *iterációs jelző* `(iteration marker)` jelzi, hogy az üzenetet fogadó objektumnak az üzenet többször is elküldésre kerül. Az iterációra általában a `*` (a több számosság jele) utal. Az UML nem határozza meg az iteráció formáját, így az lehet például `*[x>y]`, vagy `*[i:=1..n]`. Az iterációt szekvenciálisnak tekintjük, a csillagot követő két párhuzamos vonallal (`*||`) azonban párhuzamos (konkurens) iterációk is megadhatók.

üzenetátadás jellege

Az üzenet irányát a küldőtől a fogadó felé mutató nyíl jelöli. A nyílhegy az üzenetátadás jellegére utal:

- A két vonallal megrajzolt „hagyományos" nyílhegy a vezérlés egyszerű, nem függvényhívás-jellegű terjedését jelöli.
- A teljes-nyílhegy (kisméretű fekete háromszögben végződő vonal) a szinkron, függvényhívás/visszatérés jellegű üzenetváltást jelöli. Az üzenetet küldő megvárja, amíg a fogadó befejezi az üzenet feldolgozását. Párhuzamos végrehajtás esetén ezt tényleges várakozással kell megvalósítani.
- Az egyetlen vonallal rajzolt fél-nyílhegyekkel aszinkron üzenetküldést határozunk meg. Az üzenetet küldő nem várja meg az üzenet feldolgozását, hanem folytatja működését. Az aszinkron üzenet általában egy adatcsomagként megjelenő szignál („jelzés") küldését jelenti.
- Az üzenet feldolgozásából történő *visszatérést* `(return)` szaggatott vonallal ábrázoljuk. Ez nem egy speciális üzenet, hanem a feldolgozás végét és esetlegesen az eredmény értékének a visszautalását jelenti. A visszautalt értéket a visszatérés címkéjeként adjuk meg. Egyszerű szinkron üzenetek vázolása esetén gyakran nem jelöljük a visszatérést, célszerű azonban azt explicit módon ábrázolni, ha az üzenetre a hívott objektum több további üzenet küldésével reagál.

Szekvencia-diagram

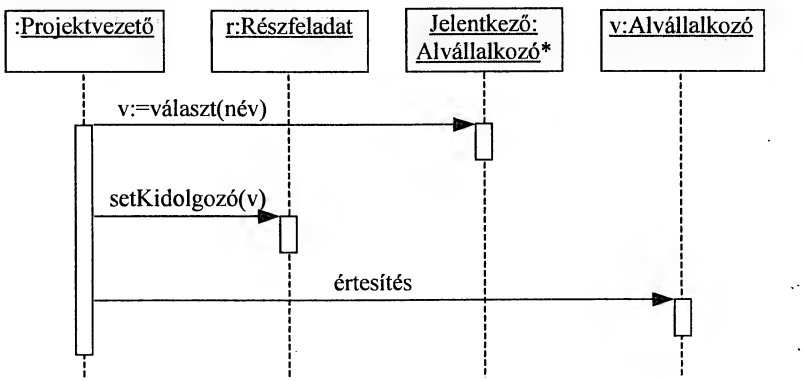
Szekvencia-diagramokkal az időbeliségük alapján ábrázoljuk az együttműködő objektumok közötti üzenetváltásokat. Konkurens folyamatok működését is vázolhatjuk, valamint megadhatunk a valós idejű rendszereknél lényeges időhatárokat. Jelölhetjük az objektum létrehozását, lebontását, az öndelegációt, illetve a különböző feltételek melletti elágazást.

A szekvencia-diagram két olyan speciális jelölést tartalmaz, amely nem található meg az együttműködési diagramokon, az életvonal és a vezérlésfókusz jelzését.

Szekvencia-diagramok `(sequence diagram)` esetén a példaobjektumokat egy-egy függőleges vonal, az *életvonal* `(lifeline)` tetején helyezzük el, amelyek az objektumok élettartamára utalnak. Az objektumokat a diagram vízszintes tengelye mentén soroljuk fel, a függőleges tengely pedig fentről lefelé az idő múlását ábrázolja.

életvonal

131. Ábra: Szekvencia-diagram



Az *üzenetküldést* `(message passing)` a küldőtől a fogadó objektum életvonaláig vezető, nyílban végződő, általában vízszintes egyenes vonal jelzi, melyen az üzenetváltás elemeit címkeként adjuk meg.

üzenetküldés jelölése

A szekvencia-diagramot a baloldalán kiegészíthetjük a folyamat rövid *szöveges leírásával*. Az objektumok függőleges vonalakkal *sávokba* `(swimlanes)` is csoportosíthatók, például a csomagjaik alapján.

sávok

A vezérlés szempontjából a létező objektum „élő" állapotán belül a következőkben három fő típust határozunk meg:

- Aktív, *működő* állapotában az objektum rendelkezik a vezérléssel, így például üzenetet küldhet egy másik objektumnak.

- *Várakozó* állapotában az objektum valamely műveletének végrehajtása közben egy elküldött üzenetének a feldolgozására vár. Ezért ebben a pillanatban nem rendelkezik a vezérléssel, de a művelet során felvett lokális változói még léteznek.
- Az objektum passzív, *felfüggesztett* állapotában is „él", léteznek az attribútumai és képes fogadni a működő objektumok üzeneteit, de éppen nem hajt végre műveletet, így lokális változói sincsenek.

vezérlés-fókusz

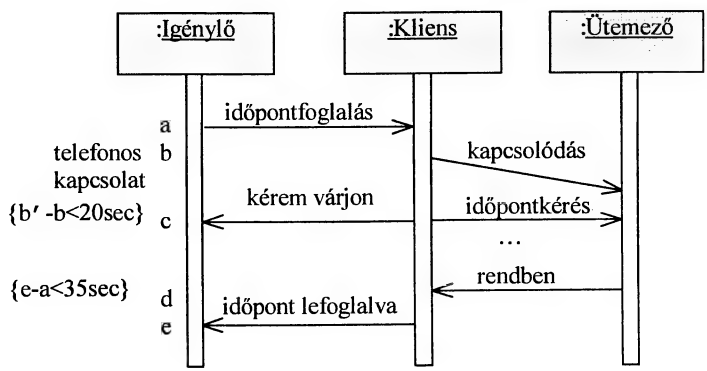
Az UML szekvencia-diagramjain a *vezérlésfókusz* (`focus of control`) jelzi a műveletet éppen végrehajtó objektumot, mely vagy működik, így a vezérléssel is rendelkezik, vagy egy általa elküldött üzenet feldolgozására vár. Mindkét esetben léteznek a művelet során felvett lokális változók.

Konkurens folyamatok

Szekvencia-diagramokkal konkurens folyamatok is modellezhetők Az üzenetátadás jellegét nyílhegyekkel adjuk meg. A teljes-nyílhegy a szinkron hívás/visszatérés, a fél-nyílhegy aszinkron üzenetváltást határoz meg, a hagyományos nyílhegy pedig egyszerűen csak a vezérlés terjedését jelöli.

aszinkron üzenetküldés

132. Ábra: Konkurens folyamatok



Az életvonal egyes időpontjai elnevezhetők a diagram baloldalára írt címkékkel. Valós idejű (`real-time`) rendszereknél lényeges, hogy a válaszok bizonyos időtartamon belül megérkezzenek. A címkék segítségével az üzenetváltások *időhatárait* (`timeout`) megszorításokként adhatjuk meg, például a `{b-a <= 2 sec}` módon rögzítjük, hogy a diagram `a` és `b` időpontja között legfeljebb két másodperc telhet el.

időhatárok

Ha egy üzenetküldést a modellezés során nem atominak, hanem időtartammal rendelkezőnek tekintünk, akkor az üzenetátadást egy olyan ferde nyíllal jelöljük, amelynél a vétel „lentebb", egy későbbi időpontban történik. Általános konvencióként az `a` címkével jelölt időpontban küldött egyetlen üzenet vételének időpontját `a'` módon jelöljük.

üzenetátadás időtartama

Üzenetek

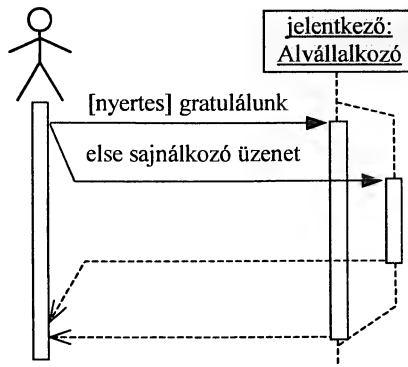
Az alapvető technikák mellett a szekvencia-diagramokon további jelölésekkel pontosíthatjuk az interakció lezajlását.

A vezérlés különböző feltételek mellett történő *elágazása* `(branch)` ugyanazon időtartamon belül a működés két alternatív lehetőségét jelenti. Szekvencia-diagramokon ezt az üzenet vonalának és az életvonal leágazásával jelöljük. A feltételek logikai értékű kifejezéseit szögletes-zárójelek között, a megfelelő üzenet *őrszemeként* `(guard)` adjuk meg.

elágazás

őrszem

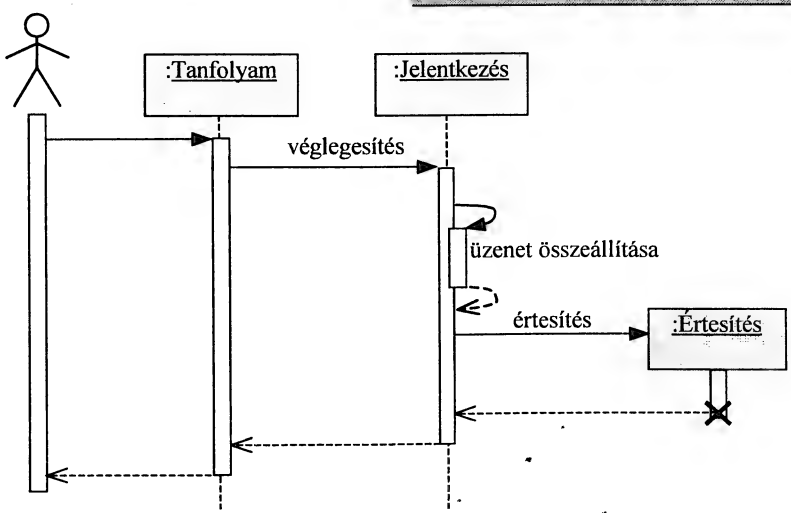
133. Ábra: Elágazás



Öndelegációnak `(self-delegation)` nevezzük, ha az objektum a saját műveletét használja fel, azaz önmagának küld üzenetet. A működésen belüli működést egy hurok jellegű nyíllal és egy kiegészítő vezérlésfókusszal jelöljük.

öndelegáció

134. Ábra: Öndelegáció és élettartam



konstrukció és destrukció

Különleges üzenet az új objektumot létrehozó konstrukció, mely nyílhegye a példaobjektumban végződik. Az objektum lebontását, destrukcióját az életvonalat lezáró X jelzi. Destrukciót kiváltó üzenet esetén a nyílhegy az X-ben végződik. A konstrukciós és destrukciós üzeneteket a «`create`», illetve «`destroy`» sztereotípiával is hangsúlyozhatjuk.

Együttműködési diagram

Az UML másik interakció-diagram típusa, az együttműködési diagram segítségével az együttműködő objektumok szerveződését, kapcsolódási módjait hangsúlyozhatjuk. Az időbeli sorrendiséget számozással jelöljük. A diagramokkal aktív és passzív objektumok konkurens működése is szemléltethető.

Az együttműködési diagramokkal nem az üzenetváltások időbeliségére, hanem az együttműködő objektumok szerveződésére, az üzeneteknek az objektumstruktúrában való terjedésére helyezzük a hangsúlyt. Az együttműködési diagramokkal részletesen ábrázolhatjuk az objektumok kapcsolódási módjait, valamint vázolhatjuk az objektumoknak az interakció során lezajló állapotváltásait is.

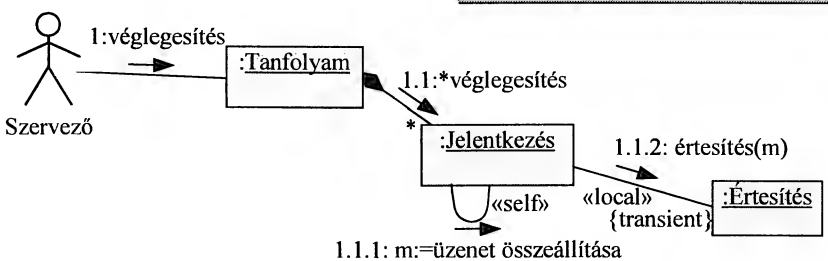
kapcsolódási módok ábrázolása

Az együttműködési diagramokon a példaobjektumok ikonokként jelennek meg, az üzenetek pedig a kapcsolatok közelében elhelyezett nyílként. A sorrendiséget az üzenetek elé írt számozás határozza meg.

A számozás lehet folytonos, vagy hierarchikus (pl.: `1.2.1`), mely utóbbival az egymásbaágyazott függvényhívások reprezentálhatók. Az *aszinkron* módon küldött *üzeneteket* az UML konvenciója szerint nem számokkal, hanem betűkkel jelöljük.

számozás

135. Ábra: Hierarchikus számozás



Az együttműködési diagramok hasonlóak az objektum-diagramokhoz, így a megfelelő sztereotípiákkal részletesen ábrázolhatjuk az objektumok kapcsolódásának jellegét. Például egy objektum a strukturális kapcsolatok mellett elérheti a lokális változóit és a paramétereit is.

Az *öndelegáció* `(self-delegation)` üzenetét az objektum önmagára mutató, «`self`» sztereotípiájú kapcsolata, egy „hurok" mellett adjuk meg. A feltételes elágazást az azonos sorszámú, de egymást kizáró feltételű őrszemekkel jelöljük.

öndelegáció

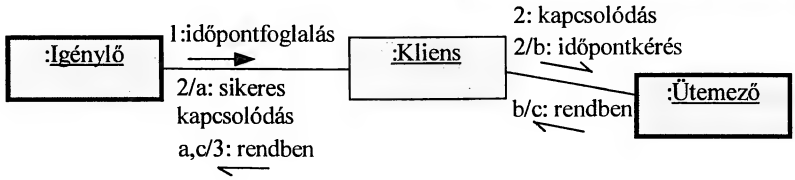
A diagramon ábrázolt interakció és az objektumok élettartamának viszonyát a kapcsolatok megfelelő végén elhelyezett `{new}`, `{destroyed}`, valamint a `{transient}` (mely a `{new destroyed}` rövidítése) megszorítással adhatjuk meg. Az objektum létrehozását és lebontását eredményező üzeneteket explicit módon is jelölhetjük a «`create`» és a «`destroy`» sztereotípiával.

élettartam

szinkronizáció

Párhuzamos végrehajtás esetén szinkronizációt a `jel1,jel2/` formában megadott előzmények felsorolásával határozhatunk meg. Az üzenet küldése mindaddig blokkolva van, amíg az adott jelű üzenetek feldolgozása be nem fejeződik. A sorszámozás is közvetve egy szinkronizációt jelöl, mivel az üzenet küldését csak akkor engedélyezzük, ha az alacsonyabb sorszámú üzenetek feldolgozása már befejeződött.

136. Ábra: Konkurens működés



Interakció-diagramok használata

Interakció-diagramok segítségével az objektumok együttes működését szemléltethetjük. Az időbeliséget hangsúlyozó szekvencia-diagramokkal és az objektumok szerveződését kiemelő együttműködési diagramokkal elsősorban a hívás/visszatérés jellegű vezérlések ábrázolhatók. Paraméterezett interakció-diagramokkal tervezési mintákat is megadhatunk.

Az interakció-diagramok az adott körülmények között együttműködő objektumok együttes viselkedését szemléltetik. Nem alkalmasak azonban az együttműködés részletes megadására.

Szekvencia-diagramokkal az interakciók időbeliségét, az üzenetváltások időbeli egymásutániságát hangsúlyozzuk, az együttműködési diagramok pedig az interakcióban érintett objektumok szerveződését, felépítését emeli ki.

A két diagramtípus ugyanazt az információt fogalmazza meg. Egy szekvencia-diagram automatikusan együttműködési diagrammá alakítható, és fordítva. A két diagramtípus közül így kiválaszthatjuk a célunknak jobban megfelelő változatot.

használati eset megvalósítása

A diagramokkal felvázolhatjuk, hogy egy használati eset megvalósítása során az érintett objektumok hogyan működnek együtt. Ekkor minden használati esethez egy- vagy több, a normál, illetve speciális végrehajtási módoknak megfelelő interakció-diagramot rendelünk. Ebben a megközelítésben a diagramok a használati esetekhez tartozó *forgatókönyvek* formalizált változatai.

Az interakció-diagramok elsősorban akkor használhatók, ha az együttműködés túlnyomóan hívás/visszatérés jellegű vezérlésen alapszik. Ekkor az interakciót egyetlen objektum kezdeményezi, illetve a működések alapvetően egy vagy több delegációra korlátozódnak.

Az interakció-diagramok bonyolult feltételek, illetve iterációk esetén áttekinthetetlenné válnak. Az értelmezhetőséget csak úgy tarthatjuk fent, ha az eltérő feltételek mellett lezajló viselkedést különálló diagramokon ábrázoljuk.

Tervezési minta

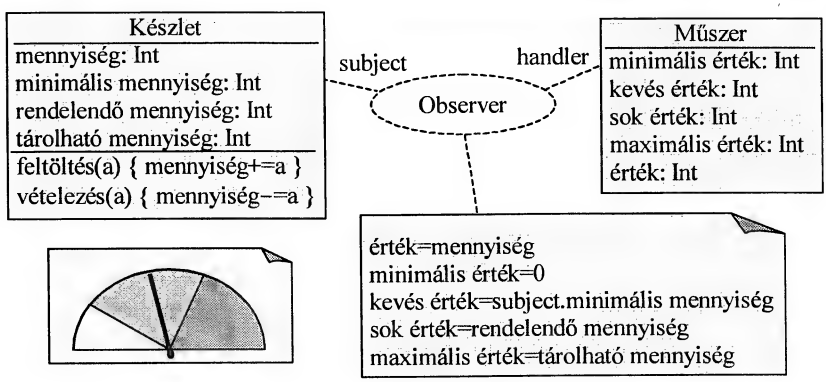
Az interakció-diagramok egy különleges felhasználási módja, ha bizonyos objektumokat paraméterekként tekintünk, amelyeket egy-egy konkrét helyzet esetén adott objektumokkal helyettesítünk. A paraméterezett interakció-diagramok egy absztrakt együttműködési módot, egy interakció-mintát írnak le, így alkalmasak *tervezési minták* (`design pattern`) formális megadására is.

tervezési minta használata

Az UML a paraméterezett interakció-diagramok felhasználását szaggatott vonallal rajzolt ellipszissel jelöli, melybe a minta nevét írjuk. A paraméterek helyére helyettesített osztályokat az ellipszishez szaggatott vonallakkal kapcsoljuk, melyeknél szerepnévként adjuk meg a megfelelő paraméter nevét.

Egy gyakran használt tervezési minta az `Observer` („megfigyelő"). Például egy készlet aktuális mennyiségét grafikusan egy olyan műszerrel szeretnénk ábrázolni, amely egy kevés, egy normál és egy „sok" tartományt tartalmaz. A készlet mennyiségének bármely változása esetén értesítenünk kell a műszert, hogy aktualizálja a mutató állását. A két objektum közötti dinamikus kapcsolat egy egyszerű megvalósítási módja az `Observer` tervezési minta alkalmazása.

137. Ábra: Tervezési minta alkalmazása



IV.

Időben lezajló változás

Az időben lezajló változást az UML két diagramtípusával ábrázolhatjuk. Aktivitás-diagramok segítségével a változást alapvetően aktív szempontból írjuk le, a végrehajtandó tevékenységek és azok sorrendiségének meghatározásával. Az aktivitás-diagramok alkalmasak munka-folyamat jellegű összetett tevékenységek, illetve konkurens működések ábrázolására. A diagramon jelölhető a tevékenységek végrehajtási helye (például objektuma), a közöttük lezajló adatforgalom, illetve a tevékenységek eredménye.

Az állapot-átmenet diagram a változást alapvetően passzív oldalról közelíti meg, a vizsgált elemet elérő külső események hatására bekövetkező reakciók, például állapot-váltások alapján. Az állapotokhoz tevékenységeket rendelhetünk, az átmenetekhez pedig akciókat, melyek eredményeként az objektum más objektumok változásába is beavatkozhat.

Aktivitás-diagramok

Aktivitás-diagramok segítségével az időben lezajló változást aktív szempontból, a végrehajtandó tevékenységek és azok sorrendiségének meghatározásával adjuk meg. A diagram-típussal párhuzamos működés, illetve szinkronizáció is ábrázolható. A tevékenységeket a sávok jelölésével végrehajtási helyekhez rendelhetjük. A tevékenységek eredménye, illetve a közöttük lezajló adatforgalom objektumfolyamként szemléltethető.

Az *aktivitás-diagram* `(activity diagram)` egy viszonylag új elem az UML diagramjainak családjában. Az eszközrendszer alapjai *munkafolyamat-diagram* `(work-flow diagram)` néven közismertek az üzleti életben, ahol azt üzleti folyamatok áttekintésére, modellezésére alkalmazzák. A jelölések másik forrása a programozás tanításának kezdetén használatos *folyamatábrák* `(flow-chart)`.

Aktivitás-diagramok segítségével az alkalmazás *dinamikáját*, *időben lezajló változását* aktív oldalról, a végrehajtandó tevékenységek sorrendiségének meghatározásával ábrázoljuk. Az UML jelölésrendszerében a dinamika szemléltetésének másik eszköze az állapot-átmenet diagram, amely alapvetően passzív oldalról, a külső események hatására történő reakciók, állapotváltások alapján közelíti meg a változást.

Az aktivitás-diagramok nagy előnye, hogy a tevékenységeknél nem csak egyszerű „egymás utáni", szekvenciális sorrendet, hanem egymás melletti, párhuzamos, *konkurens* végrehajtást is meghatározhatunk. A diagramok alkalmasak több szereplő által végrehajtott munkafolyamat-jellegű tevékenységek, illetve párhuzamos folyamatok ábrázolására.

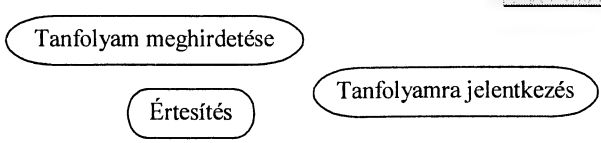
Aktivitás és sorrendiség

tevékenység

Az aktivitás-diagramok alapeleme maga az *aktivitás*, vagy más néven *tevékenység* `(activity)`. Az aktivitás egyszerűen egy olyan feladat, amit meg kell csinálni. Az aktivitás az alkalmazásban egy osztály valamely műveletének hívásaként, „használataként" jelenik meg vagy egyszerűbb esetben egy műveleten belüli egy vagy több utasításként.

Az aktivitás jele az íveltoldalú téglalap, a „lóversenypálya".

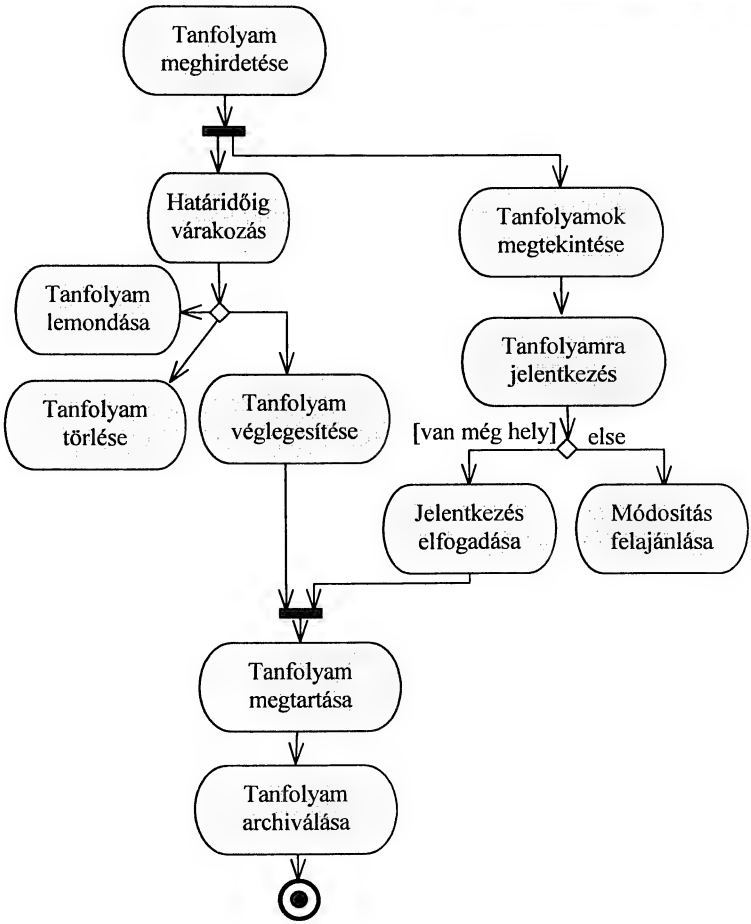
138. Ábra: Aktivitás



A végrehajtandó tevékenységek időbeli sorrendjét a nyílhegyben végződő vonallal, az *átmenettel* `(transition)` jelöljük. A nyíl rendre az aktivitás befejezése/befejeződése utáni következő végrehajtandó tevékenységre mutat, így egyszerű *szekvenciális* `(sequential)` sorrendet határoz meg.

átmenet

139. Ábra: Aktivitás-diagram



Ha bizonyos tevékenységcsoportok esetén nem lényeges a sorrend, azok egymás mellett, konkurensen is végrehajthatók, akkor a *szinkronizációs vonal* `(synchronization bar)` segítségével a vezérlés több *szálra* `(thread)` bontható.

szinkronizációs vonal

A szinkronizációs vonal jelölése egy vastag, vízszintes vonal.

A szálakkal nem feltétlenül párhuzamos végrehajtást írunk elő, csupán jelezzük, hogy a tevékenységcsoportok logikailag függetlenek egymástól. Ha nem szükséges tényleges párhuzamosság, akkor a szálak tevékenységcsoportjai egyszerűen egymás után, de tetszőleges sorrendben végrehajthatók.

szinkronizáció

Ugyancsak a szinkronizációs vonallal jelöljük a *szinkronizációt* `(synchronization)`, azt a pontot, ahol a konkurensen végrehajtható szálak befejeződését megvárjuk. A szinkronizációval jelezzük, hogy a következő tevékenység végrehajtásához szükséges az előző aktivitások sikeres végrehajtása.

Ha a szálakat egyszerűen, szinkronizációs-vonal nélkül egyesítjük, akkor úgy tekintjük, hogy az első befejeződő szál megszakítja a többi szálat és a vezérlés a megfelelő irányba halad tovább.

vezérlés

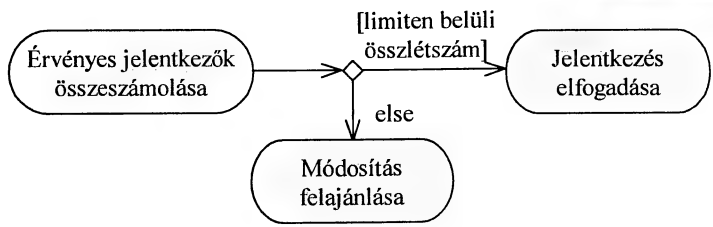
Az átmeneteknél az interakció-diagramoknál megismert két vezérlés is megadható:

- A * jelöli az iterációt, amely alkalmas például a „minden egyes elemre" feldolgozások megadására. Szinkronizációs vonal után megadott iteráció aktivitásai az egyes elemekre párhuzamosan hajtódnak végre.
- A szinkronizációs vonalhoz írható őrszem, azaz feltétel, amely megadható szövegesen vagy formálisan, például egy programozási nyelv kifejezéseként. Minden beérkező szál esetén a feltétel ellenőrzésre kerül és teljesülése esetén a vezérlés a kimenőszálon folyatódik, a többi ág pedig félbeszakad. A * segítségével hangsúlyozhatjuk, hogy minden párhuzamosan indított iterációs szál befejeződését meg kell várni. Ugyanez feltételként is megadható (pl.: `[minden elem ellenőrizve]`). A * az alapértelmezés, ezért csak akkor alkalmazzuk, ha a szinkronizációra fel szeretnénk hívni a figyelmet, mivel a feltétel hiánya azt jelzi, hogy minden befutó szál befejezte működését.

döntés

Az alternatív tevékenységcsoportok közötti választást, a leágazást a *döntés* `(decision)` szimbólumával, egy rombusszal jelöljük. A döntésből kivezető éleket szögletes zárójelek között megadott feltételekkel, „őrszemekkel" címkézzük.

140. Ábra: Döntés

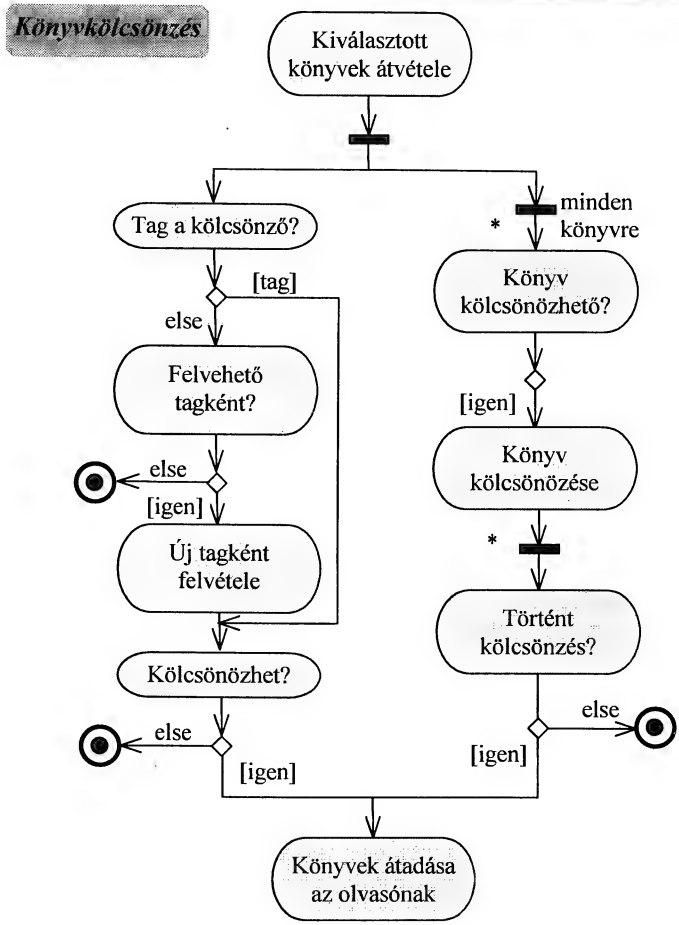


A szinkronizációs vonal „és", a döntés „vagy" jellegű elágazást jelöl.

Szinkronizációs vonal esetén mindegyik szálat, azaz egyik „és" a másik „és" a harmadik... tevékenységcsoportot végre kell hajtani. A vezérlés ágai függetlenek, így ugyanabban az időben, párhuzamosan is végrehajthatók.

Döntés esetén a feltételtől függően „vagy" az egyik, „vagy" a másik, „vagy" a harmadik... ág hajtódik végre, így egy időben a felsoroltak közül csak egy tevékenységcsoport hajtódik végre.

141. Ábra: Szinkronizáció és döntés



Akció- és aktivitás-állapot

Az UML definíciója szerint az aktivitás-diagramokon a végrehajtandó műveleteket *állapotokként* `(state)` vehetjük fel, az íveltoldalú téglalap így az állapot jelzésére szolgál. Az UML az állapotokat két fő típusra, akció- és aktivitás-állapotok csoportjára bontja.

Akció-állapotban `(action state)` akciókat helyezhetünk el. *Akciónak* `(action)` az atomi műveletet nevezzük. Úgy tekintjük, hogy egy atomi művelet végrehajtása nem igényel időt, így az egyetlen időpillanatban zajlik le és ezért félbe sem szakítható. Akció például egy attribútum beállítása egy kifejezésből számított értékre, vagy egy művelet végén a visszatérési érték visszautalása. Ugyancsak akciónak tekintjük az objektum létrehozását és törlését, valamint a jelzés `(signal)` küldését.

akció-állapot

aktivitás-állapot

A műveletek másik csoportját az *aktivitások*, vagy más néven a *tevékenységek (activity)* alkotják, amelyeket úgy tekintünk, hogy a végrehajtásuk időt vesz igénybe, ezért félbe is szakíthatók. Az *aktivitás-állapotban (activity state)* egy vagy több aktivitás szerepelhet. Az aktivitás egy, végső soron atomi akciókból felépített összetett tevékenységet jelöl, amelyet egy másik aktivitás-diagramon részletesen ábrázolhatunk. Ugyanakkor egy aktivitás-diagram egyetlen összetett tevékenységként is tekinthető, így az egy másik diagramon egyetlen aktivitás-állapotként jelölhető.

Az aktivitás-állapotban még további elemek is megadhatók, például be- és kimeneti akciók, melyeket részletesen az állapot-átmenet diagramok esetén tárgyalunk.

A diagramon két különleges, úgynevezett *pszeudo-állapot (pseudo state)* is szerepelhet.

kezdőállapot

A fekete körrel (körlappal) jelölt *start-állapot (start state)*, más néven *kezdőállapot (initial state)* határozza meg az összetett tevékenység kezdetét. Egy diagramon több kezdőállapot is lehet, melyet több párhuzamos szál indításaként értelmezünk.

záróállapot

Körön (körvonalon) belüli fekete körrel *(bull eye)* jelöljük a *stop-állapotot (stop state)*, más néven a *záróállapotot (final state)*, mely az összetett tevékenység befejeződését jelöli.

összetett tevékenység

Az aktivitás-diagramon megfogalmazott összetett tevékenység alapvetően két esetben érhet véget.

- Ha minden működő szál elér egy olyan aktivitás-állapotot, amelyből nem vezet ki további él, azaz nincs több végrehajtandó tevékenység, akkor az összetett aktivitás *befejeződik*.
- Ha valamelyik szál elér egy zárószimbólumot, amely a tevékenység végét határozza meg, akkor az összetett tevékenység is befejeződik úgy, hogy a többi szál működése *félbeszakad*.

Aktivitás és objektum

Az egyszerű aktivitás-diagram csak a végrehajtandó tevékenységeket és azok időbeli sorrendiségét határozza meg, de nem jelöli, hogy a műveletet ki hajtja végre, illetve azt, hogy a tevékenységet milyen adatáramlás kíséri végig.

A megadott akció- és aktivitás-állapotok a megfelelő aktorok vagy osztályok műveleteiként vagy azokon belüli utasításokként, illetve kifejezésekként jelennek meg.

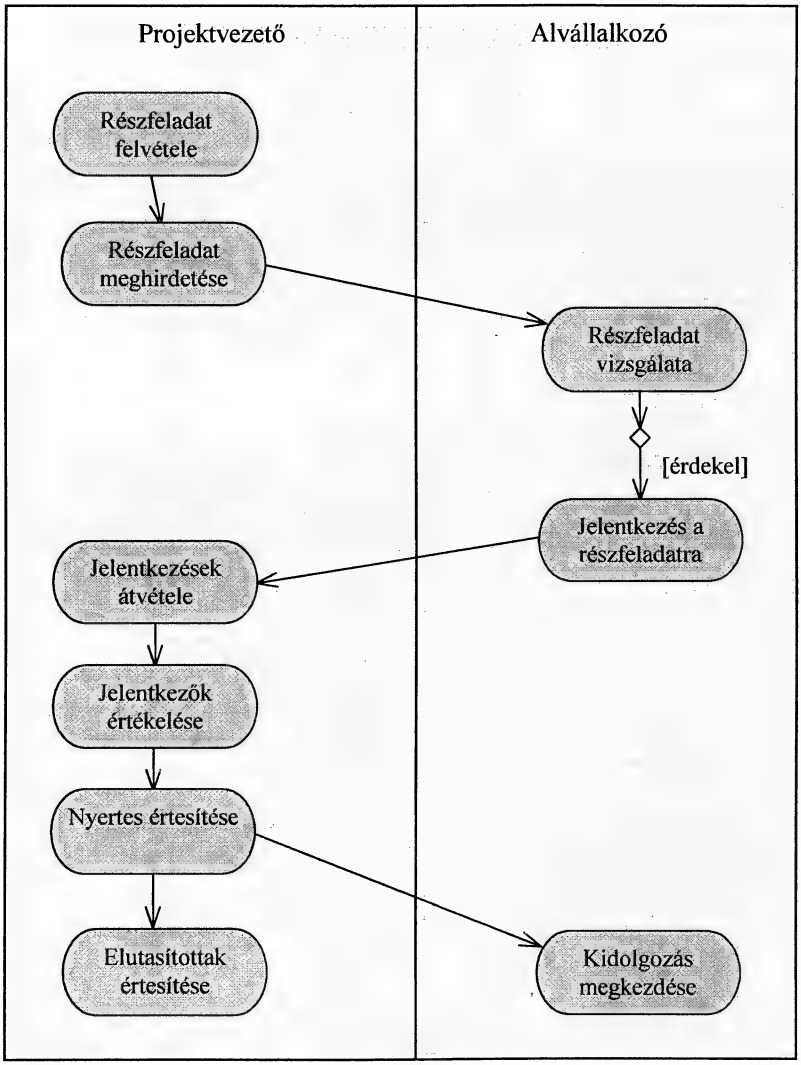
sávok

A tevékenységek végrehajtóit az UML jelölésrendszerében a *sávokkal (swimlanes)* ábrázoljuk, amellyel a tevékenységeket végrehajtási helyekhez rendeljük, képszerűen: „húzzuk szét". Az eredeti *sávok (swimlanes)*, szó szerint: „úszósávok" kifejezésre legtöbbünk lelki szemei előtt valószínűleg egy úszóverseny jelenik meg. A sávokat vonalakkal vagy téglalapokkal jelenítjük meg, melyek tetejére írjuk a végrehajtási helyek (a „versenyzők") nevét.

Az UML az aktivitás-diagramokat elsősorban az üzleti folyamatok modellezésére ajánlja, ezért a sávokhoz nem rendel különösebb szemantikát.

A ***Q*** modellező módszerben a sávokat aktoroként vagy példaobjektumokként tekintjük. Ebben az értelmezésben a sávok segítségével ábrázoljuk az összetett tevékenység elemeinek aktorokhoz, illetve osztályokhoz rendelését.

142. Ábra: Sávok



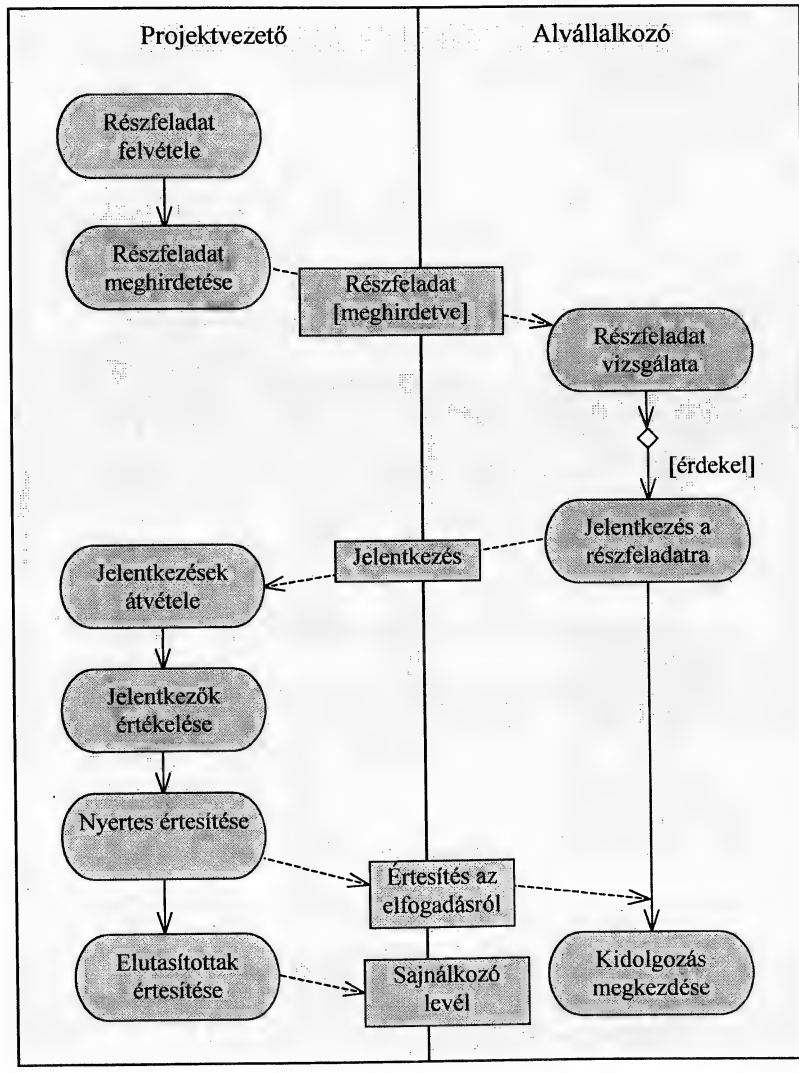
Gyakori, hogy az aktivitások között a végrehajtás sorrendjét meghatározó vezérlésnél szorosabb kapcsolat is létezik. Egy munkafolyamatot általában dokumentumok kísérnek végig, illetve több tevékenységet bizonyos objektumon hajtanak végre.

objektum-folyam

Az UML a tevékenység eredményének megadására az *objektum-folyam* `(object-flow)` jelölését vezeti be. A tevékenység eredménye egy objektum, melybe az állapotból kiinduló szaggatott vonalú nyíl vezet. Az objektumból kiinduló és egy tevékenységben végződő szaggatott vonalú nyíl jelöli, hogy a művelet milyen adatokat, azaz objektumokat használ fel. Ha egy tevékenység egy objektumot átad egy másik tevékenységnek, akkor azt az objektummal és egy kettős nyíllal ábrázoljuk.

Mivel egy objektum többször is szerepelhet egy diagramon, illetve annak az attribútumértékei és az állapota is megadható, így ábrázolhatjuk, hogy az egyes tevékenységek az objektumon milyen módosítást hajtanak végre.

143. Ábra: Objektum-folyam



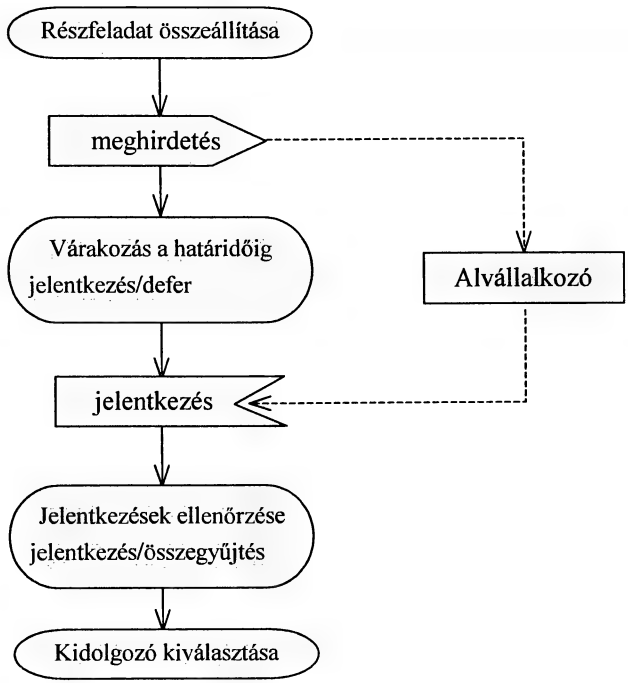
Szignál küldése és fogadása

Az összetett tevékenység végrehajtása közben szükséges lehet, hogy bizonyos külső elemeknek jelzést küldjünk vagy azoktól jelzést fogadjunk. A *jelzés* `(signal)` lehet egy utasítás vagy a tevékenység közben fellépő helyzet jelentése. *jelzés*

Az UML a jelzés küldését egy téglalap és az oldalához illesztett háromszög („hegy") kombinációjával jelöli. A külső elemtől érkező jelzés fogadásának alakzata (a „zászló") esetén a téglalapból „kivágjuk" a jelzésre utaló háromszöget. A fogadás ábrázolásával azt is hangsúlyozzuk, hogy a jelzésig várakozunk és csak annak megérkezése után vesszük a következő tevékenységet.

A küldő, illetve fogadó objektumot az alakzathoz a jelzés terjedésének megfelelő irányú szaggatott vonalú nyíllal kapcsoljuk.

144. Ábra: Szignálküldés, késleltetés és fogadás



Bizonyos jelzésekre egy tevékenységen belül, annak megszakítása nélkül is válaszolhatunk. Ezt a változatot *belső akcióként* `(internal action)` adjuk meg, ahol az állapoton belül a jelzés elnevezését követő / jel után írjuk a jelzés hatására végrehajtandó akciót, egy időtartam nélkülinek tekintett atomi műveletet. A jelzés feldolgozása nem szakítja félbe a tevékenységet, ugyanakkor nem is kell megvárni annak bekövetkeztét. *belső akció*

Az UML értelmezésében a feldolgozatlan jelzés elveszik. Előfordulhat azonban, hogy egy eseményt mégsem a bekövetkeztekor akarunk feldolgozni, hanem „késleltetjük" bizonyos tevékenységek végrehajtásakor.

késleltetett jelzés

A *késleltetett jelzést* a tevékenységen belül az esemény neve után írt `defer` speciális művelettel adjuk meg. Ha a következő tevékenységre lépünk át, akkor az eseményt fel kell dolgozni vagy újból késleltetni kell, különben az esemény elveszik.

Állapotdiagram

Az állapot-átmenet diagramok az időben lezajló változást alapvetően passzív módon, a külső hatásokra történő reakciók alapján közelítik meg. A reakciók általában átmenetekként, az objektum állapotának megváltozásaként jelennek meg. Az állapotokhoz tevékenységeket rendelhetünk, az átmenetekhez pedig akciókat.

Az UML-ben használatos *állapot-átmenet diagram* `(state-transition diagram)` vagy röviden *állapotdiagram* `(state diagram, state chart)` lényegében a David Harel által kidolgozott jelölésrendszert jelenti, amelyen néhány, az objektumorientált technikák követelményeinek megfelelő kisebb módosítást és kiegészítést tettek.

Az aktivitás-diagram mellett az állapotdiagram az UML másik eszköze az időbeli változások ábrázolására. Az aktivitás-diagram alapvetően aktív módon, a végrehajtandó tevékenységek alapján írja le a dinamikát. Ezzel szemben az állapotdiagram egy passzív szempontot képvisel, mivel a változást az események hatására történő reakciók, állapotváltások segítségével fogalmazza meg.

Állapot és átmenet

állapot

Egy kiválasztott időpillanatban adott osztály objektumát teljességében az *állapota* `(state)` segítségével jellemezhetünk. Az egyértelműbb szóhasználat miatt az állapottal kapcsolatos további fogalmakat is bevezetünk.

- Az objektum *belső állapotának* nevezzük attribútumainak az összességében, együttesen tekintett értékeit.
- Az objektum *külső állapota* az objektum által a vizsgált időpontban elért, ahhoz közvetlenül vagy közvetve kapcsolódó összes objektum együttes belső állapota.
- Az objektum *kiterjesztett állapota* az adott időpont belső és a kapcsolatai által meghatározott külső állapota.

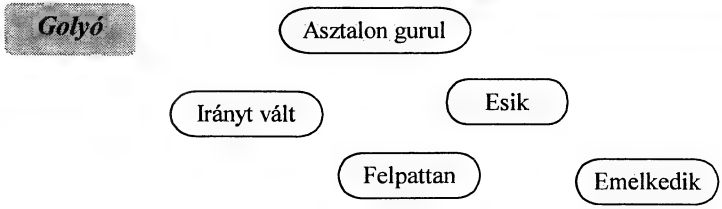
Állapoton a következőkben kiterjesztett állapotot értünk.

konkrét és absztrakt állapot

Az objektum adott helyzetben történő jellemzéséhez általában nem szükséges az állapotot teljes részletességében ismerni, mivel a helyzet szempontjából lényegtelen részletektől egyszerűen eltekinthetünk. Az összes konkrét attribútumérték által meghatározott *konkrét állapot* `(concrete state)` helyett a legtöbbször elegendő az objektumot egy magasabb absztrakciós szinten vázlatosan, az *absztrakt állapotával* `(abstract state)` leírni. Egy absztrakt állapot több konkrét állapot lehetőségét (matematikailag: azok halmazát), azaz egy *állapot-jelleget* foglal össze egyetlen definícióban. Például a `Telefon` objektumnak egy absztrakt állapota a tárcsázás, függetlenül attól, hogy eddig milyen számjegyeket ütöttünk le.

Állapoton a következőkben absztrakt vagy konkrét állapotot értünk.

145. Ábra: Absztrakt állapot



állapottal történő jellemzés

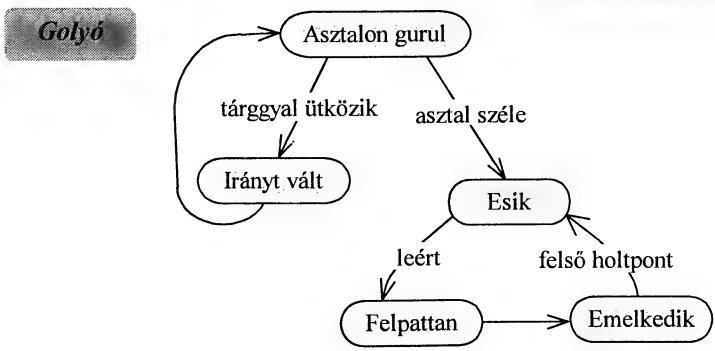
Az állapottal történő jellemzés az objektumot (illetve az ahhoz kapcsolódó többi objektumot) egyetlen egységnek tekinti, míg az attribútumok értékei az állapotot elemeire darabolják fel. Az állapot meghatározása így egy magasabb szempontból vett, áttekintőbb képet ad az objektumról.

A valós világ objektumainak modellezése esetén általában nem is szükséges minden részletet ábrázolnunk. Az állapotok segítségével megfogalmazhatjuk a valóban lényeges jellemzőket, így egy absztraktabb képet alkothatunk az objektumról, melyet a tervezés során attribútumokként, illetve kapcsolatokként valósítunk meg.

dinamika: állapot változása

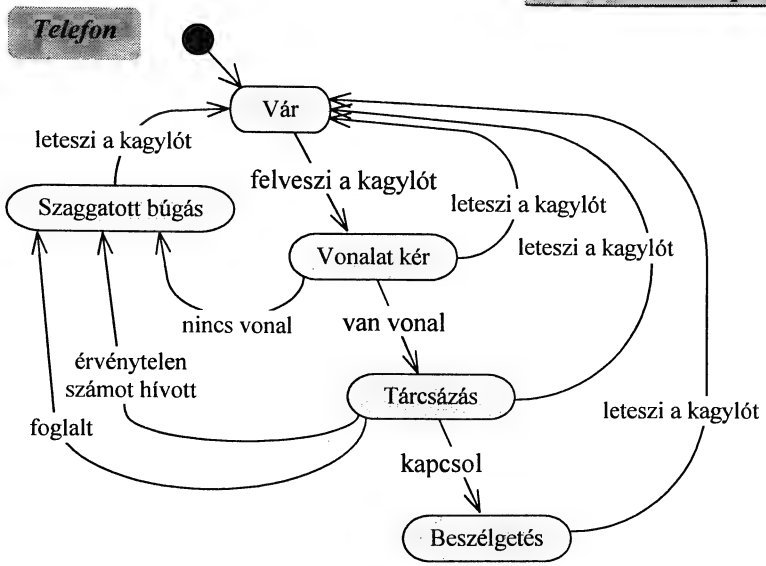
Nem konstans objektumnak változhatnak az attribútumértékei és a kapcsolatai, azaz az állapota. Az *időben lezajló változást*, az objektum *dinamikáját* ezért az állapot változásával írhatjuk le. A változás elemi egysége az adott állapotból egy másikba történő állapotváltás, amelyet *átmenetnek* `(transition)` nevezünk. Az átmenetet legtöbbször valamely *esemény* `(event)` váltja ki, így a külső változásra történő reakciót állapotváltásként fogalmazhatjuk meg.

146. Ábra: A változás



Az állapot-átmenet diagramon megadjuk az objektum lehetséges állapotait, és az azokat összekapcsoló átmeneteket, az állapotok lehetséges változásait. Az átmenet bekövetkezhet az objektumot elérő esemény hatására, melyet az átmenet címkéjeként jelölünk.

147. Ábra: Állapotdiagram



Úgy tekintjük, hogy átmenet esetén az objektum „átugrik" egyik állapotából a másikba. Az átmenetek így atomi egységek, félbeszakíthatatlanok („oszthatatlanok"), melyek egyetlen időpillanat alatt zajlanak le. Az átmenet tehát egyetlen időponthoz kapcsolható.

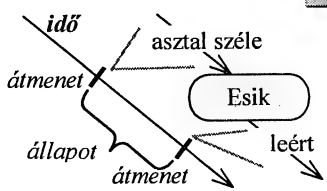
állapot: atomi

Ezzel szemben az állapot két különböző időpontot jelöl, amikor az objektum átlép az állapotba, és amikor elhagyja azt. Az állapot ezért azt az időtartamot is jelenti, amíg az objektum az adott állapotban van, például absztrakt állapot esetén a konkrét állapot változása még a jellegen belül marad. Az állapot időtartamhoz kapcsolható, így az félbe is szakadhat, például egy esemény hatására.

átmenet: időtartam

Az objektum változásában az átmenetek jelentik a mérföldköveket, a változás pillanatait. Az objektum bármely két különböző átmenete, két egymásutáni mérföldkő pedig egy állapotot határoz meg, amely a két időpont közötti időtartamon belül felvett konkrét állapotokat csoportosítja.

148. Ábra: Állapot és átmenet



Az asztalon guruló, tárggyal ütköző, majd leeső és pattogó golyó esetén tőlünk függ, hogy az irányváltást és a felpattanást időtartammal rendelkező állapotnak vagy egyetlen időpillanatban lezajló eseménynek tekintjük. A diagramon az állapotként történő modellezést választottuk, például azért, mert feltételezzük, hogy a golyó rugalmas anyagból készült, így az ütközés és a felpattanás időt vesz igénybe.

A valós világ objektumai automatikusan, egymással való kölcsönhatásaik eredményeként változnak. Az alkalmazást számítógépen szeretnénk megvalósítani, ami alapvetően algoritmikus működésű, azaz segítségével egy meghatározott cél elérésének a módja fogalmazható meg. Ezért a külvilág objektumainak automatikus egymásra hatását szándékosságra, *vezérlésre* kell átváltanunk. Az asztal lapján guruló golyót modellezve az asztalnak „üzennie" kell a golyó-objektumnak, hogy az elérte az asztal szélét, ezért a golyónak a földre kell esni.

Esemény

Valamely esemény az objektum állapotának megváltozását eredményezheti, melyet az állapotdiagramon úgy jelölünk, hogy az átmenetet az eseménnyel címkézzük meg.

lényegi történés

Az UML definíciójában az *esemény* `(event)` egy olyan *lényegi történés*, amelynek adott helye és időpontja van. A „lényegi" `(significant)` jelzőt úgy értelmezzük, hogy egy esemény bizonyos objektumok megváltozását eredményezheti. Az esemény időponthoz kötődik, ezért úgy tekintjük, hogy nincs időtartama. Az állapotokhoz hasonlóan általában nem egyedi, hanem *absztrakt eseményeket* adunk meg úgy, hogy a hasonló jellegű eseményeket egyetlen csoportba soroljuk.

Az UML négyféle eseményt határoz meg. Az első két típust összefoglaló néven üzenetnek nevezhetjük. Az objektumorientált szemléletben az *üzenet* `(message)` két objektum közötti információ-cserét, azaz egy interakciót jelent. Üzenet a következő lehet:

hívás-esemény

- A *hívás-esemény* `(call event)` egy szinkron interakciót jelöl. Ekkor az üzenetet küldő objektum megvárja az üzenet feldolgozását. A hívás-esemény megvalósítható úgy, hogy az üzenetet küldő objektum meghívja a fogadó objektumnak az üzenet nevével és paramétereivel azonos műveletét, melyben a fogadó objektum az üzenet feldolgozásának lépéseit határozza meg. A fogadó objektum a feldolgozás eredményét általában visszatérési értékként utalja vissza.

jelzés

- *Jelzés* vagy *szignál* `(signal)` *küldésével* `(send)` aszinkron interakciót határozhatunk meg. Ebben az esetben az üzenetet küldő nem várja meg az üzenet feldolgozását, hanem folytathatja a tevékenységét, bár a szignál hatására a működése félbe is szakadhat. A jelzésre példa a *kivétel* `(exception)`, amellyel a hibát okozó kivételes körülményekről értesítjük az adott műveletet meghívó „külső" műveleteket. Jelzés esetén az üzenet feldolgozásának eredményét csak egy újabb üzenet küldésével közölhetjük.

Az üzenetnek lehetnek paraméterei, így a definiálása és az üzenet küldése a műveletekkel azonos módon történik. Az UML a konkrét szignált olyan üzenetként tekinti, amelyet objektumként valósítunk meg. A diagramokon a szignál típusát egy `«signal»` sztereotípiájú osztállyal definiáljuk.

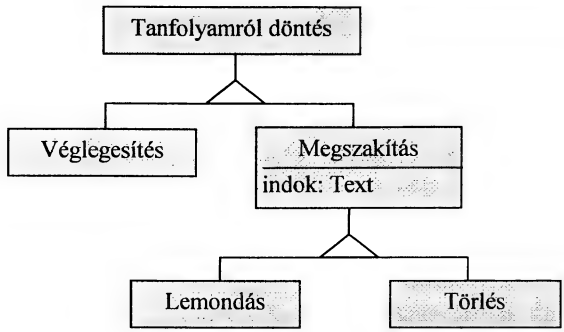
Nem csak a szignál, hanem a hívás esemény is modellezhető osztályként. Ekkor az üzenet neve az osztály nevének felel meg, a paraméterek pedig attribútumokként

jelennek meg. Az esemény előfordulásának idejét, valamint a helyét (az üzenetet küldő forrás-objektumot) általában az esemény attribútumának tekintjük, különösen szignálok esetén. Az üzenethez megadhatunk műveleteket is, melyek például a „paraméterek" elérését teszik lehetővé, segítségükkel visszaüzenhetünk a küldőnek vagy jelezhetjük az üzenet vételét.

üzenetek öröklési hierarchiája

Az osztályként történő modellezés segítségével az üzenetek öröklési hierarchiába rendezhetők. Az absztrakt üzenetekből további attribútumok, esetleg műveletek felvételével pontosabb üzeneteket származtathatunk (például: egérgomb lenyomása — jobb egérgomb lenyomása).

149. Ábra: Események származtatási hierarchiája



Az UML által definiált másik két esemény az idő és a változás esemény.

idő-esemény

- Az *idő-esemény* `(time event)` valójában egy *időhatár* túllépését `(timeout)` jelenti. Kiköthetjük, hogy az objektum egy állapotban csak egy előre megadott időtartamot tölthet és annak elteltével automatikusan egy átmenet következik be. Az idő-eseményt az `after` szó után írt, időtartamra vonatkozó kifejezéssel adjuk meg, például `after 10 ms`, `after(1 seconds)` vagy `after 24 hours a jelentkezés után`.

változás-esemény

- A *változás-esemény* `(change event)` a paraméterként megadott helyzet állapoton belüli bekövetkezését jelenti. Változás-eseményt a `when` szó után írt logikai kifejezéssel, feltétellel adhatunk meg, például `when amount>maxAmount`, vagy `when(n+i>100)`. Változás-esemény egy abszolút időpont is lehet, például `when(11:45 p.m.)`, amely az aktuális időre vonatkozó vizsgálatot rövidít: `when(time=11:45 p.m.)`.

Idő-eseményt általában valamely időzítő mechanizmussal valósítunk meg. A változás-esemény a programban folytonos tesztként jelenik meg, amely teljesülését az állapoton belül, a működés bizonyos pontjain ismételten ellenőrzünk.

Esemény és átmenet

őrszem

Átmenethez az azt kiváltó esemény mellett egy feltétel, más néven *őrszem* *(condition, guard)* is kapcsolható (például: „`[nincs jelentkező]`"). Az őrszem csak akkor engedélyezi az átmenetet, ha a rá bízott feltétel teljesül. A feltétel maga is az objektum kiterjesztett (absztrakt vagy konkrét) állapotára vonatkozik, melybe az esemény paramétereit is beleértjük. Az őrszem feltételét szögletes zárójelek között az esemény neve után írjuk, mellyel azt jelöljük, hogy a feltétel teljesülését az esemény bekövetkeztekor kell megvizsgálni.

átmenet akciói

Az átmenethez műveleteket is megadhatunk. Mivel az átmenet egyetlen időpontban zajlik le, a művelet csak atomi, időtartam nélküli *akció* *(action)* lehet, tevékenység azonban nem. Az átmenet esetén végrehajtandó akciókat perjel (/) után sorolhatjuk fel, például `/teljesLétszám+=jelentkezés.létszám`.

Az átmenetként megjelenő változásról az objektum egy újabb üzenetben, egy általa generált eseménnyel tájékoztathat más objektumokat. Az üzenetküldést ugyancsak akciónak tekintjük, melyet egy hívás esemény esetén a művelet hívásával, szignál küldésekor pedig a `send` segítségével (például `send Túl sok jelentkező`) adunk meg.

automatikus átmenet

Gyakran egy állapot egyetlen célja, hogy egy tevékenységet végrehajtson, majd annak befejeztével az objektumnak egy másik állapotba kell átváltani. Az esemény nélküli, címkézetlen állapotváltás az *automatikus átmenetet* *(automatic transition, default transition)* jelöli. Ekkor az objektum az állapothoz rendelt tevékenységet befejezve automatikusan a következő állapotba vált át.

Foglaljuk össze, hogy egy átmenetet milyen elemekkel adhatunk meg:

- Átmenetet kiváltó *esemény*, melynek paraméterei („attribútumai") is lehetnek. Ha nem szerepel, akkor ez az automatikus átmenetet jelöli.
- *Őrszem*: szögletes zárójelek között az átmenet feltétele.
- Átmenet esetén végrehajtandó *akciók*, melyeket / után sorolhatunk fel.

Állapot

A gráfként tekintett állapotdiagramban a csomópontokban helyezzük el az objektum állapotait, melyeket a lehetséges átmenetek élei kapcsolnak össze. A diagramon két kitüntetett, más néven *pszeudo-állapot* *(pseudo state)* lehet.

kezdőállapot

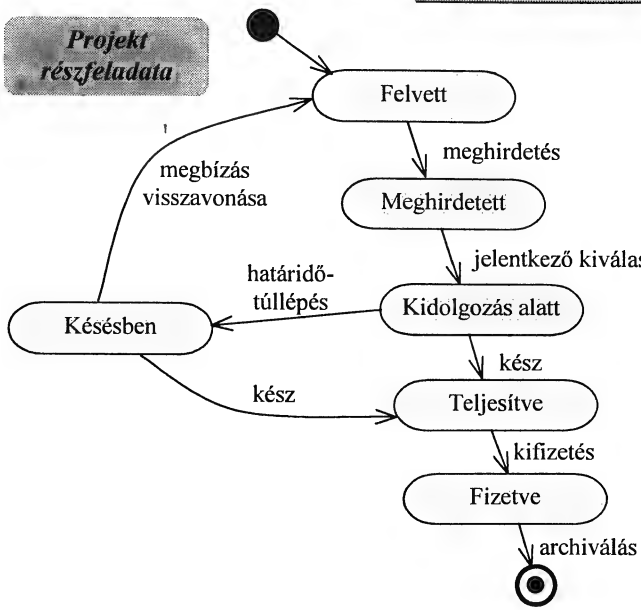
záróállapot

- Készítésekor az objektum automatikusan a *kezdőállapotába* *(initial state, start state)* lép; ebbe nem vezethet él.
- A *záróállapot* *(final state, stop state)* elérése az objektum lebontását eredményezi, ezért ebből nem vezet ki további él.

„élő" állapot

Az objektumnak alapvetően két, „élő", illetve „nem élő" állapota lehetséges. A kezdőt követő és a zárószimbólumot megelőző átmenet két időpillanata jelenti az objektum élettartamát *(life cycle)*. Minden felvett állapot valójában az objektum élő állapotának a pontosítása.

150. Ábra: Kezdő- és záróállapot



Ha a diagramon létezik záróállapot, akkor azt *egy-életciklusú diagramnak* `(one-shot diagram)` nevezzük, mivel adott körülmények az objektum megszűnését eredményezhetik. Ez nem következhet be, ha a diagramban nem létezik záróállapot, így azt *végtelen ciklusú diagramnak* `(continuous loop)` nevezzük.

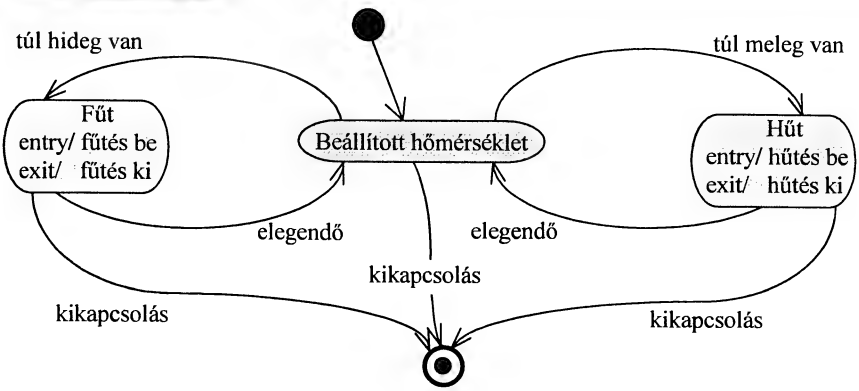
egy-életciklusú és végtelen ciklusú diagram

Az állapot alakzatán belül az állapot nevét, valamint különböző műveleteket adhatunk meg:

állapot elemei

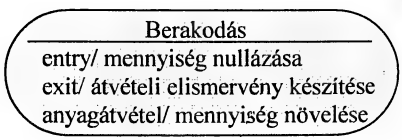
- Az alakzat felső részébe írjuk az *állapot nevét*. Névtelen, anonymus állapot esetén nem adunk meg nevet.
- Az állapot időtartammal rendelkezik, így az állapothoz *tevékenységek* `(activity)` is megadhatók, az állapot neve alatt, `do/` után. A tevékenység is rendelkezik időtartammal, ezért egy esemény hatására félbeszakadhat. Azt is egy speciális állapotnak tekintjük, amíg az objektum a tevékenység végrehajtásával van elfoglalva. Több tevékenységet a végrehajtási sorrendjüknek megfelelően sorolunk fel.
- Az állapothoz *bemeneti akció* `(entry action)` rendelhető, melyet `entry/` jelölés után adunk meg. A bemeneti akció az állapotba történő átváltás (és így a tevékenység) előtt hajtódik végre. Azt az esetet rövidíti, amikor az állapotba vezető minden átmenet esetén azonos tevékenységet akarunk végrehajtani.
- Az állapothoz *kimeneti akció* `(exit action)` is rendelhető, melyet `exit/` jelölés után adunk meg. A kimeneti akció az állapotból történő kilépés után hajtódik végre. Azt az esetet rövidíti, amikor az állapotból kivezető minden átmenet esetén azonos tevékenységet akarunk végrehajtani.

151. Ábra: Be- és kimeneti akciók



- Az állapot *belső akciója* `(internal action)` esetén a megadott esemény akciókat (pontosabban reakciókat) vált ki anélkül, hogy az állapot megváltozna. Az állapotnál az esemény elnevezését és esetleges paramétereit követő / jel után soroljuk fel a végrehajtandó akciók neveit. Olyan átmenetnek a rövidítése, amikor az állapotból kiinduló, a megfelelő eseménnyel címkézett él ugyanabba az állapotba tér vissza, de ekkor nem hajtódik végre sem kimeneti, sem bemeneti akció, mivel az objektum ugyanabban az állapotban marad.

152. Ábra: Belső akció



Az állapothoz tartozó tevékenységet félbeszakíthatja egy külső esemény, azonban a kimeneti akció ekkor is végrehajtódik. Akció nem szakítható félbe, mivel az nem rendelkezik időtartammal.

Üzenetküldés

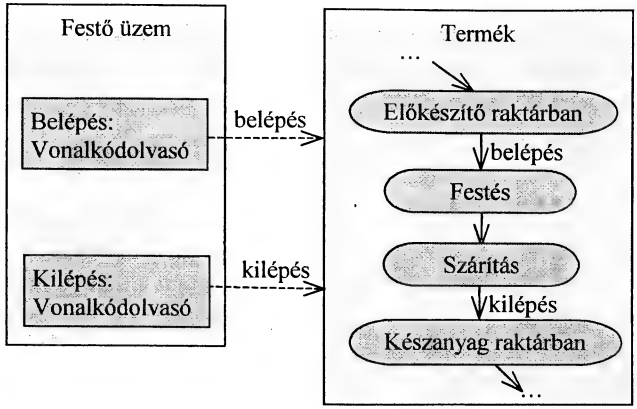
Az objektum állapotának megváltozása kihathat más objektumokra, azokban is változást idézve elő. Ennek egy megvalósítási lehetősége, ha az objektum egy üzenetben értesíti az érintett objektumokat.

üzenetküldés vagy esemény-generálás

Az *üzenetküldést*, vagy más néven *eseménygenerálást* szövegesen a fogadó objektum(ok) és az üzenet segítségével adjuk meg, mely utóbbi a művelet hívásával azonos formájú: `fogadóObjektum.üzenetNév(paraméterek)`.

Grafikusan az üzenetküldést a fogadó objektumban vagy multiobjektumban végződő szaggatott vonalú nyíl jelöli, melyet a küldött üzenettel címkézünk meg.

153. Ábra: Üzenetküldés



Szignál küldését az üzenetküldés elé írt `send` kulcsszóval jelöljük. A «`send`» sztereotípiájú függőséget alkalmazzuk, ha jelezni szeretnénk, hogy a függő osztály generálhat a használt «`signal`» osztálynak megfelelő szignált.

szignál küldése

Az üzenetküldést az átmenet jelzésének, arra vonatkozó tudósításnak tekintjük, így az átmenethez hasonlóan ez is egy atomi művelet, egy akció. Az üzenetküldést az állapotok határain adhatjuk meg: az átmeneteknél, illetve be- és kimeneti akcióként.

Tevékenység és akció

Gyakran egy állapot egyetlen célja, hogy valamely műveletet végrehajtson, majd annak befejeztével az objektum automatikusan a következő állapotba váltson át. Az állapotdiagram műveleteit *(`operation`)* két fő típusba sorolhatjuk:

- A *tevékenység (`activity`)* végrehajtása időt vesz igénybe, ezért ez félbeszakítható, illetve az állapothoz kapcsolható.
- Az *akció (`action`)* olyan atomi, azaz félbeszakíthatatlan művelet, melynek eltekintünk a végrehajtási idejétől. Az akció az átmenethez kapcsolható.

Ha az állapot mindössze egy művelet végrehajtására korlátozódik, akkor azt rövidítve egy tevékenység-, illetve akció-állapotként adhatjuk meg.

Az *akció-állapot (`action state`)* önmagában kissé ellentmondásos fogalma esetén az állapot csak akciókat tartalmaz. Ez az „állapot" nem félbeszakítható, mivel valójában egyetlen átmenetet jelöl, így csak be- és kimeneti akciókkal rendelkezhet, belső akciókkal és tevékenységgel azonban nem.

akció-állapot

A *aktivitás-állapot (`activity state`)* alapvetően egy tevékenység végrehajtását jelöli, amely félbeszakítható. A tevékenység több elemi műveletből épül fel, melyeket külön diagramon is ábrázolhatunk.

aktivitás-állapot

Az akció- és a tevékenységállapotot általában nem nevezzük el, hanem a névrészben a műveletet adjuk meg.

Az UML mindkét speciális állapot jelölésére ugyanazt az alakzatot használja, így a művelet alapján kell meghatároznunk, hogy az melyik típusba tartozik.

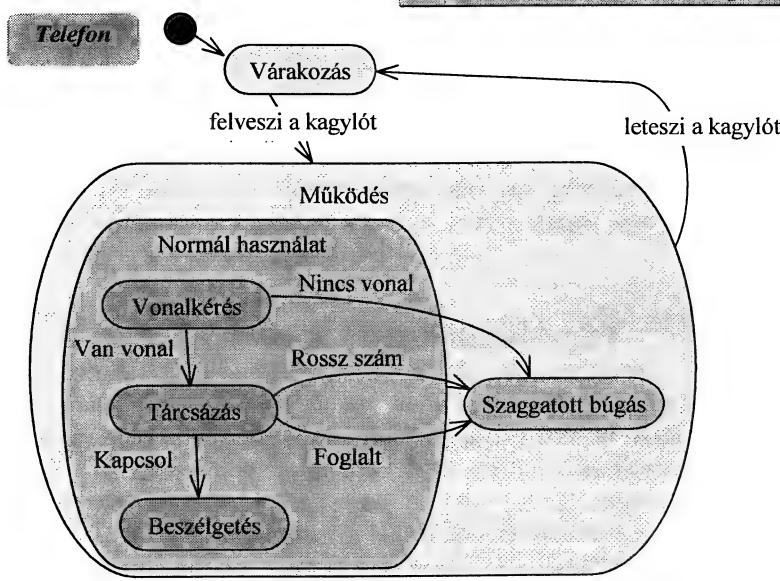
Strukturált állapotdiagram

Összetett viselkedésű objektum esetén a fenti eszközökkel csak nehezen áttekinthető diagramokat rajzolhatunk. Nagyszámú állapot és átmenet esetén az állapotokat követhetetlenül sok él köti össze, ami túlságosan bonyolult diagramot eredményez.

strukturálás

Az áttekinthetőséget az állapotdiagram strukturálásával tarthatjuk fent. Az elemibb állapotokat *szuperállapotokba* csoportosíthatjuk, egy vázlatos állapotot pedig az alállapotaira bonthatunk.

154. Ábra: Strukturált állapotdiagram



szuper-állapot

A több alállapotot magában foglaló *szuper-* vagy *absztrakt állapot* több, időben egymást követő alállapotot (egész-rész irány), vagy több speciális állapot vázlatos megfogalmazását (általánosítás-pontosítás) is jelentheti. Az absztrakt állapotok az osztályokhoz, illetve az eseményekhez hasonlóan öröklési hierarchiába rendezhetők. Az alállapotok öröklik a szuperállapot átmeneteit, ha azt nem definiálják át. Egy `Telefon` objektum esetén a `Működés` szuperállapot `Tárcsázás` és `Beszélgetés` alállapotai egyaránt öröklik azt az átmenetet, hogy ha a kagylót letesszük, akkor a telefon `Várakozás` állapotba vált át. A szuperállapot egy átmenete az összes alállapot azonos átmeneteit rövidíti.

A strukturálásra két technikát használunk. A szuperállapothoz tartozó alállapotokat vagy egy nagy ívelt oldalú téglalapon belül *csoportosítjuk*, vagy azt egy *külön állapotdiagramon* fejtjük ki.

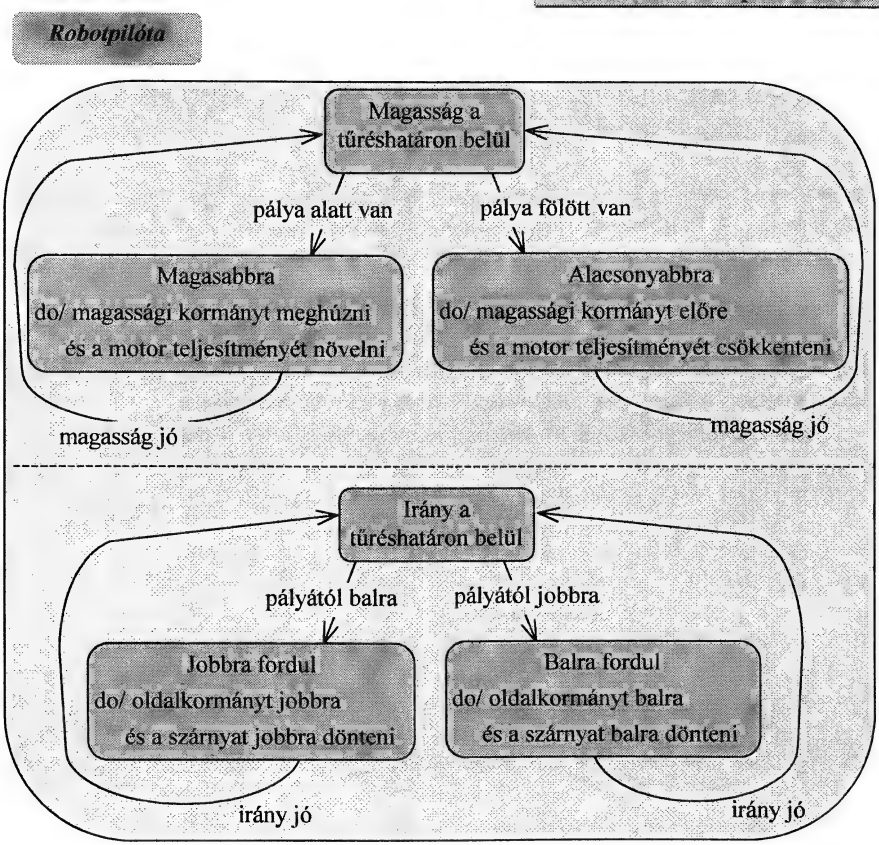
Konkurencia

A valóság objektumai az időben egyszerre léteznek és változnak. Együttes, párhuzamos működés esetén egy időpillanatban a teljes rendszer állapota a részek együttes állapotával írható le. Ez egy kompozíció-jellegű „és" kapcsolat, mivel az együttes állapotot az egyik részállapot és a másik részállapot és... közösen adják meg. Minden egyes rész állapotváltását külön aldiagramon ábrázolhatjuk.

objektumon belüli konkurencia

Objektumon belüli konkurencia esetén az összetett objektum attribútumai és kapcsolatai önállóan változó részhalmazokra bonthatók. Az objektumon belüli konkurencia jelölésére olyan absztrakt állapotot használunk, amelyet pontozott vonallal kettő vagy több részre osztunk fel. Minden rész önálló kezdőállapottal rendelkezik. A kompozit állapot határát elhagyó átmenet az összes többi konkurens működést félbeszakítja, és az objektum az átmenet által meghatározott következő állapotba vált át.

155. Ábra: Állapot és átmenet



Az aktivitásdiagramoktól kissé eltérően a kompozit állapot automatikus átmenete akkor következik be, ha az összes belső állapotváltás eléri a zárószimbólumot.

V.

Adatfolyam-diagram

Kiegészítésül áttekintjük a korábbi rendszerszervezési módszerekben elterjedten használt adatfolyam-diagramok jelöléseit. Az adatfolyam-diagram nem része az UML eszközrendszerének, mégis jól alkalmazható bonyolult számítási folyamatok szemléltetésére, valamint a rendszeren belüli adatforgalom ábrázolására.

Adatfolyam-diagram

A hagyományos strukturált, illetve a korábbi objektumorientált módszerekben elterjedten alkalmazott adatfolyam-diagramot az UML már nem vette fel eszközkészletébe. Adatfolyam-diagram segítségével jól ábrázolható egy bonyolultabb számítás felépítése, valamint alkalmas a rendszeren belüli adatforgalom szemléltetésére.

A hagyományos strukturált rendszerszervezési módszerekben az adatfolyam-diagramok még alapvető jelentőségűek voltak, mivel azok segítségével lehetett megadni a feldolgozás elemeit és a közöttük lévő adatforgalmat. A korábbi objektumorientált módszerekben, például az OMT-ben is még helyet kapott ez a diagramtípus, de a jelentőségét fokozatosan elvesztette. A jelölésekkel többnyire a szervezeteken belüli adatforgalmat szemléltették, de konkrét fejlesztésekben csak elenyésző mértékben alkalmazták.

A diagramtípus háttérbe szorulásának egyik oka, hogy alapvetően nem objektumorientált szemléletű. Az OO világban a feldolgozást már többnyire az objektumok együttműködéseként valósítják meg, nem pedig egy összetett transzformációként. Talán ez az oka, hogy az UML sem vette át ezeket a jelöléseket, valamint az, hogy az aktivitás- és az interakciós diagramokkal is részben ábrázolható az adatáramlás.

használata

Kiegészítésként mégis áttekintjük az adatfolyam-diagramok jelöléseit, és ennek az az oka, hogy segítségükkel jól szemléltethetők az összetett transzformációkból felépülő műveletek belső szerkezete. A diagramok jól használhatók:

- A rendszeren belüli adatáramlás ábrázolására,
- Függvény-jellegű, összetett (például több tucat függvényből felépülő) számítások megadására.

A jelölések ismertetésének másik oka, hogy az adatfolyam-diagramok összekapcsolhatók az objektumorientált szemlélettel (erre maga a Booch-módszer is példát mutatott), és így egy nagyon hatékony eszköz birtokába jutunk, mellyel a *folytonos transzformációk* (`continuous transformation`) jól szemléltethetők.

A harmadik ok pedig az, hogy a közeljövőben valószínűleg előtérbe kerülnek a valós világ összetett hatásait a folytonos transzformációk alapján modellező rendszerek. Ezt a módszert alkalmazzák például egy változó rendszert valós időben síkban (2D) vagy térben (3D) megjelenítő alkalmazások esetén, valamint szimulációk és játékprogramok készítésekor.

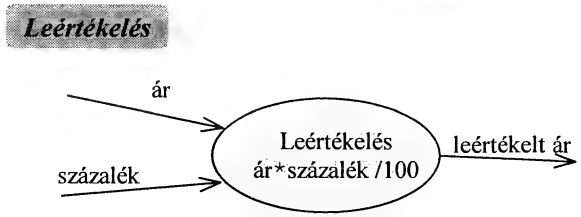
transzformációk megadása

Adatfolyam-diagram esetén eltekintünk az időtől. Olyan transzformációt, származtatási módot adunk meg, amely a bemeneti értékeket a diagramon meghatározott számítási módszerrel a megfelelő kimeneti értékekké alakítja át. A diagram ezért akciók és megszorítások ábrázolására is alkalmas.

Feldolgozás és adatfolyam

Az *adatfolyam-diagram* `(data flow diagram)` az értékek függését ábrázolja. A diagram alapelemei a feldolgozások, melyeket az adatfolyamokat reprezentáló élek kapcsolnak össze.

156. Ábra: Adatfolyam-diagram

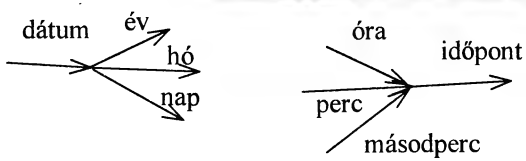


A *feldolgozás* `(process)` a bemeneti értékeket a kimeneti értékekké transzformálja. A feldolgozás jele az ellipszis alakzat, melybe a művelet definícióját írjuk. A feldolgozás el is nevezhető. *feldolgozás*

A feldolgozásban végződő élek a feldolgozás *bemenetét* (inputját), az abból kivezető élek pedig az eredményt, a *kimenetet* (outputot) jelölik. A feldolgozás így a bemeneti és kimeneti adatok mintáját („szignatúráját"), valamint a transzformáció módját határozza meg. A feldolgozás megadásának egyértelműnek kell lenni, de azt tetszőleges eszközzel megfogalmazhatjuk, például matematikai függvényként, egyenlettel vagy egyszerűen szöveges leírással. *be- és kimenet*

Az *adatfolyamot* `(data flow)` a diagram nyílban végződő irányított élei jelölik, melyeket címkékkel is azonosíthatunk. A címkén az adat elnevezése mellett a típusa is szerepelhet. Az adatfolyam egy feldolgozás kimenetét egy másik feldolgozás bemenetére is kapcsolhatja, azaz ekkor a számítás köztes (közbülső) értékét jelöli. A diagram határain túlról érkező él a teljes adatfolyam-diagram bemenete, a határt elhagyó él pedig a kimenet. Egyetlen érték a „villával", *leágazással* `(fork)`, azaz a nyílban végződő élből kiinduló újabb élekkel egyszerre több helyre is küldhető, így a köztes adat egyszerre több helyen is felhasználható. *adatfolyam* *leágazás*

157. Ábra: „Villa" és összetett érték képzése



Egyetlen adatfolyam *összetett értéket* `(aggregate value)` is továbbíthat. Az összetett adatból olyan leágazással képezhetünk részeket, melyek mellé címkeként a részt azonosító nevet írjuk. Hasonló módon egyes részadatok az adatfolyamok egyesítésével egyetlen összetett adattá kapcsolhatók.

Aktív és passzív objektumok

aktor

Adatfolyam-diagramok esetén *aktornak* (`actor`) olyan aktív objektumot nevezünk, amely adatok termelésével vagy a kiszámított adatok kérésével és felhasználásával vezérli a feldolgozást. Az aktorok a diagram határain helyezkednek el, ezért azokat gyakran *határolóknak* vagy *lezáróknak* (terminátoroknak) is nevezik. Az aktort a diagramon téglalappal jelöljük, amelyből kiindulhatnak, vagy amelybe érkezhetnek adatok; ezek a teljes diagram bemenetei és kimenetei.

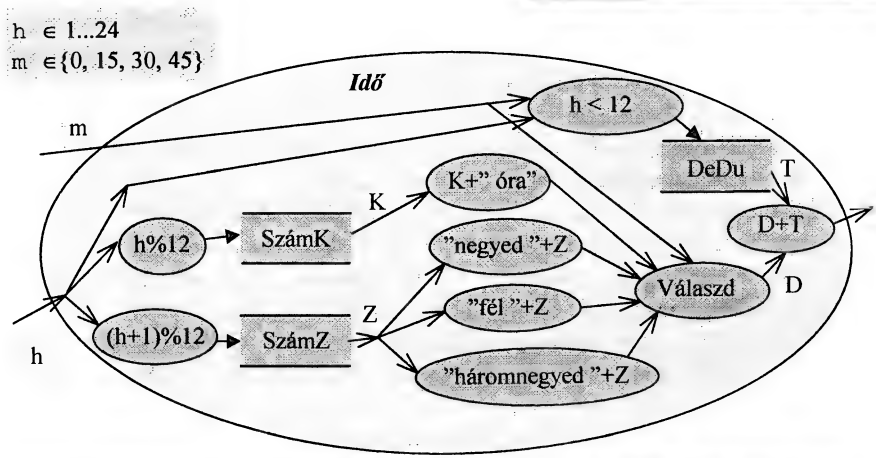
adattár

Az *adattár* (`data store`) az adatfolyam-diagram olyan passzív objektuma, amely az adatokat a későbbi elérés céljából megőrizni képes. Az adattár lehetővé teszi, hogy az értékeket a generálásuktól eltérő sorrendben is elérhessük. Az adattárat két vízszintes vonallal jelöljük, melyek közé az elnevezést írjuk. Az adattárba irányuló adatfolyam az adat tárolását, az abból kiinduló él pedig az abban tárolt adat elérését és felhasználását jelenti.

Az adattár összetett értéket, illetve adott módon (lista, tábla, adatbázis) több értéket is tárolhat — ebben az esetben *összetett (kompozit) adattárról* beszélünk. Az összetett adattár rendelkezik a beszúrás, törlés, módosítás, illetve a keresés és kiválasztás műveleteivel is. *Konstans adattárnak* nincs bemenete, az abban szerepelő értékek elérhetők, de nem módosíthatók.

konstans adattár

158. Ábra: Konstans adattárak




```
SzámK:  { „tizenkét",  „egy", „két"... }
SzámZ:  { „tizenkettő", „egy", „kettő"... }
DeDu:   { „délelőtt", „délután" }
```

A példa a Java programnyelv jelöléseit használja.

Az adattárakat általában objektumokként implementáljuk, de célszerű azokat megkülönböztetni más objektumoktól. Az adattárakat csak speciális módon használjuk, azoknak mindössze néhány speciális művelettel kell (elérés, stb.) rendelkezniük.

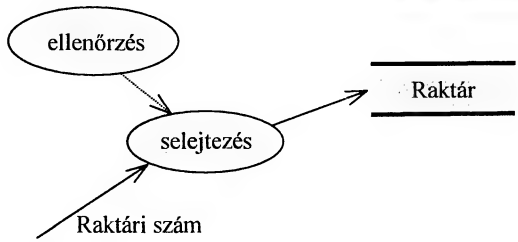
Vezérlésfolyam és strukturálás

A *vezérlésfolyam* `(control flow)` olyan logikai érték, amely meghatározza, hogy a feldolgozást végre kell-e hajtani. Az adatfolyam-diagram eltekint az időtől, így nem adja meg, hogy a számítások milyen sorrendben hajtódnak végre — ezek az állapot-, illetve aktivitásdiagramokkal rajzolhatók meg. Néha azonban valamely végrehajtás feltételét célszerű beépíteni az adatfolyam-diagramba.

vezérlés-folyam

A vezérlésfolyam a logikai értéket kiszámító feldolgozásból kiinduló pontozott vonal, amely a vezérelt feldolgozás határán nyílban végződik. A pontozott vonal mellé írjuk a végrehajtás feltételét.

159. Ábra: Vezérlésfolyam



Az adatfolyam-diagram is *strukturálható*, így összetett számításokat célszerű kisebb funkciókra bontani. Egy diagramot felbonthatunk több, egymással csak lazán kapcsolódó részre, illetve egy vázlatot pontosabb aldiagramokkal is ábrázolhatunk.

strukturálás

Az adatfolyam-diagram megrajzolását célszerű úgy kezdeni, hogy annak meghatározzuk a be- és kimeneti értékeit, azt egyetlen feldolgozással ábrázoljuk, majd ezután azt elemibb eljárásokra bontjuk mindaddig, amíg az egyszerű függvények szintjéig eljutunk.

VI.

Reprezentáció

Ebben a részben áttekintünk néhány, a modellelemek reprezentációjával kapcsolatos technikát. Az osztály adatokat és műveleteket is tartalmaz, ezért szinte minden modellelem reprezentálható osztályként, például önálló művelet, üzenet, esemény és az állapot is.

Az objektumorientált környezetek a származtatást automatikusan csak a legegyszerűbb változatban valósítják meg, így a bonyolultabb eseteket a modell átstrukturálásával kell megoldanunk. Az implementáció során az egyszerű pontosítás és az aggregáció azonos módon jelenik, ezért szükség esetén egy származtatást aggregációként reprezentálhatunk, vagy fordítva.

Az állapotdiagramok állapotai logikai értékű származtatott attribútumokként, az eseményeket pedig műveletekként tekinthetjük. Egy állapotdiagramot így a megfelelő műveletek elő- és utófeltételeként értelmezhetjük, melyek általában egzaktul megadják az objektumnak a művelet hatására bekövetkező változását

Osztályként történő reprezentáció

Az osztály egyszerre tartalmaz adatokat, valamint műveleteket, ezért szinte minden modellelem reprezentálható osztályként. Osztály segítségével megvalósíthatunk önálló műveletet. A szolgáltató osztályok más objektumokon egy műveletcsoport végrehajtását teszik lehetővé. A felhasználói felületeket egyszerűen a szakterületi modell elemeihez kapcsolhatjuk egy különleges „tükrözési" technika felhasználásával.

osztály univerzalitása

Az objektumorientált szemlélet *osztály* fogalma egyszerre tartalmazza az adatok struktúráját és azok elérésének, kezelésének műveleteit. Az osztály így rendelkezik bizonyos *univerzalitással*, mivel az alkalmazáson belül gyakorlatilag bármi megvalósítható osztályként.

értékosztály

Az entitás egyik legjellemzőbb tulajdonsága az egyediség. Ha eltekintünk az egyediségtől, az attribútumokat pedig egyszerűen értékelemeknek tekintjük, akkor az osztály alkalmas az *összetett értékek* reprezentációjára.

művelet

Egy `p(a:A, b:B)`, visszatérési értékkel nem rendelkező *művelet* ugyancsak megvalósítható osztályként. Ekkor definiálnunk kell egy `p` osztályt, amely rendelkezik az eredeti művelet lépéseit tartalmazó `◂ (a:A, b:B)` konstruktorral. A művelet egyes belső változóit az osztály attribútumaiként is felvehetjük, a részműveleteket pedig metódusokként adjuk meg. Ha a `p(a:A): R` műveletnek van visszatérési értéke, akkor az osztályban vegyünk fel egy `R()` konverziós műveletet, így a „hívást" a `p(aVal).toR()` módon valósítjuk meg.

szolgáltató osztály

horgony

A ***Q*** módszer különleges elemei a *szolgáltató osztályok* `(service class)`, melyek egy vagy több osztályon adott művelet végrehajtását kísérik végig vagy adott műveletcsoport használatát teszik lehetővé. A szolgáltató osztály a konstrukció során az alaposztályokhoz speciális módon, a *„horgonnyal"* `(anchor)` kapcsolódik, mely egy gyenge kapcsolat és lehetővé teszi az alaposztály védett `(protected)` elemeinek is az elérését.

A szolgáltató osztály végrehajthat egyetlen, a konstrukcióban megadott műveletet, annak eredményéről pedig a megfelelő konverziós művelettel kérhetünk tájékoztatást. A szolgáltató osztály arra is alkalmas, hogy segítségével elérhetővé tegyünk egy vagy több objektumon végrehajtható műveleteket. Alkalmas például arra, hogy egy objektum egy másik objektum számára bizonyos funkcióit elérhetővé tegye. A technika a visszafelé mutató hivatkozás `(backward reference)` speciális változata, amelyet átnyújtott szolgáltatásnak `(backward service)` is nevezhetünk.

A szolgáltatás befejeztével az objektum megszűnik és a „horgonyok" is lekapcsolódnak.

üzenetküldés

Az objektumok közötti információátadást, az üzenetküldést általában a hívott osztály műveleteként valósítjuk meg. Korábban megvizsgáltuk, hogy az üzenetek osztályokként modellezhetők, például öröklési hierarchiába is rendezhetők. Egy üzenet azonban nem csak modellezhető, hanem reprezentálható is osztályként. Az `m(a:A,b:B)` üzenet esetén definiálnunk kell az `m` elnevezésű és megfelelő konstruktorú osztályt. Az üzenet paramétereit általában az osztály attribútumaiként tároljuk el. Az osztály műveletekkel is rendelkezhet, melyek például az attribútumok elérését teszik lehetővé. Az „üzenetet" reprezentálhatjuk szolgáltató osztályként is, így a fogadó objektum ismerheti az üzenet forrását (`getSource()`), illetve a szolgáltatásokon keresztül el is érheti azt.

Osztályként reprezentált üzenet esetén a fogadónak rendelkeznie kell egy speciális, például `receive` vagy `notify` elnevezésű művelettel, amely alkalmas egy vagy több üzenettípus vételére. Az elnevezést kiegészíthetjük az üzenet típusával (pl.: `notifyAction(action: Action)`), vagy a műveletet különböző paraméterű változatokban is definiálhatjuk (például: `receive(m:MouseEvent)`, `receive(m:KeyboardEvent)`).

Általánosabb és így könnyebben újrafelhasználható elemekből építhetjük fel a programunkat, ha az objektum más objektumoknak közvetve is tud üzenetet küldeni. A közvetettséget úgy valósíthatjuk meg, hogy az objektum a változásának bizonyos pontjain „jelenti" a helyzetét, azaz eseményt generál, mely eljut az objektumhoz bejelentkező *figyelőkhöz (`listener`)*.

helyzetjelentés

A *helyzetjelentés* részben hasonló az UML *jelzés (`signal`)* fogalmához. A helyzetet reprezentáló *eseményt* elnevezzük és annak paramétereket is adhatunk, pl.: `event(a:A, b:B)`. A helyzet jelentése, más néven az esemény generálása történhet művelettel (`fireEvent(a:A, b:B)`). A másik megvalósítási mód egy `Event` elnevezésű és megfelelő konstruktorral és attribútumokkal rendelkező osztály és az alaposztály egy új műveletének a kombinációja (`fire(e :Event)`, vagy `fireEvent(e :Event)`). A műveletek feladata, hogy a jelentést eljuttassa a figyelőkhöz.

Az alaposztálynak nyilván kell tartania a hozzá bejelentkezett figyelőket. A nyilvántartás általában halmaz vagy rendezett halmaz jellegű módon történik, például az

```
addEventListener(l:EventListener)
removeEventListener(l:Listener)
everyEventListener(): Every<EventListener>
```

módon.

A figyelők értesítése azoknak egy adott művelete segítségével történik, mely művelet a jelzést vagy paraméterként, vagy egy osztályba csomagolva kapja meg. Adott esemény vagy eseménycsoport figyelésének műveleteit általában interfészként határozzuk meg, például:

```
«interface» EventListener:
   notify (e :Event)
```

Esetenként nem csak „előre", hanem egy üzenet feldolgozásából „hátra", a hívók felé is szeretnénk üzenetet küldeni. Általában erre akkor van szükség, ha valamely különleges eseményről, például egy hibáról szeretnénk tájékoztatni a hívót.

kivételek

A *kivételek* `(exception)` segítségével lehetőségünk van arra, hogy a műveletet a normálnak tekintett végrehajtásnak megfelelően fogalmazzuk meg, amelyet kiegészítünk a kivételes helyzetek jelzésével. Sajnos a jelenlegi programnyelvi implementációk a kivétel jelzésekor a műveletet félbe is szakítják, így csak nehezen oldható meg, hogy a jelzést követően maga a művelet hajtson végre bizonyos helyreállító `(recover)` tevékenységeket.

A kivételt általában osztályként valósítjuk meg, melyet a programnyelv megfelelő utasításával „dobunk" el, pl.: `throw new EmptyStack()`. A kivételek általában szolgáltató osztályok, melyekkel elérhetjük a kivételt generáló objektumot.

Az „előre", illetve „hátra" küldött helyzetjelentések össze is kapcsolhatók, így egy osztály egy műveleten (`fireEmptyStack()`) keresztül jelzi a helyzetet, mely eljuthat a figyelő objektumokhoz. Ha azok nem kezelték le az üzenetet, akkor a művelet egy kivétellel értesítheti a hívókat.

állapotok

A műveletekhez, illetve az üzenetekhez hasonlóan az *állapotok* is reprezentálhatók osztályokként. Az `S` állapotot például megvalósíthatjuk egy azonos nevű osztályként. Ekkor az állapot be- és kimeneti tevékenységének az osztály konstruktora és destruktora felel meg, a belső akciókat pedig műveletekként adhatjuk meg. Az állapoton belüli tevékenységet egy külön, például `Do` elnevezésű műveletként vehetjük fel, a belső változókat pedig attribútumokként jelöljük.

A reprezentáció azért lehetséges, mert az osztály maga is az objektum legáltalánosabb „élő" állapotát jelöli. A konstruktor és a destruktor az élő állapot be- és kimeneti akciói, bizonyos műveletek pedig az élő állapoton belül végrehajtható belső akciók.

Felhasználói felületek

osztályok és csomagok

A ***Q*** modellező módszer az osztályokat és a csomagokat `(package)` egységesen kezeli. A követelmények és a használati esetek mellett a csomagoknak is lehetnek attribútumaik és műveleteik. Másrészt osztályokra is megfogalmazhatunk követelményeket és használati eseteket, valamint egy osztályon belül újabb osztályokat is definiálhatunk. A módszer a csomagot a régi modul fogalomhoz hasonlóan egyetlen fordítási egységként kezeli, így a csomag annyival különbözik egy osztálytól, hogy a csomagon belüli tartalmat egyszerre kell lefordítani. Jegyezzük meg, hogy az UML értelmezésében a csomag mindössze a csoportosítás eszköze, így attribútumaik és műveleteik sem lehetnek.

A módszer esetén a csomagoknak általában (de nem mindig) osztályokat is megfeleltetünk, melyek a teljes csomagra vonatkozó attribútumokat és műveleteket tartalmazzák. Az egyszerűbb használati eseteket az osztályokként reprezentált csomagok műveleteiként vehetjük fel.

A **Q** módszer a *felhasználói felületeket* osztályokként reprezentált állapotoknak tekinti. A felületek állapot-jellegűek, mivel azok egy adott időponttól adott időpontig jelennek meg. Az UML jelöléseivel azonban nem jelölhető az állapotok származtatása, illetve kompozíciója. Ugyanakkor az állapotok osztályokként is reprezentálhatók. A felületeket ezért egy olyan osztálydiagramon ábrázoljuk, amelyen a függőségekként megadott átmenetek a felületek közötti navigációt jelölik. A felületeknek így megadható a kompozíciója és a pontosítása is. A felhasználói felületek adatait ugyanakkor hiba lenne a diagramon attribútumokként felvenni, mivel áttekinthetőbb és szemléletesebb képet kapunk, ha a felületeket eredeti megjelenési formájukban ábrázoljuk.

A felhasználói felületek különleges jellege abból ered, hogy azok a belső rendszer kivetített képeit állítják elő, illetve lehetőséget adnak arra, hogy segítségükkel módosításokat végezhessünk el.

A **Q** módszer ajánlása szerint a felhasználói felületeket és a szakterületi modell elemeit egy *tükrözésnek* `(mirroring)` nevezhető technikával kapcsoljuk össze. A tükrözés lényege az, hogy a felületeken megjelenítjük adott objektumok alap- és származtatott attribútumait, azokat beállíthatjuk, illetve a segítségükkel az objektumokon műveleteket hajthatunk végre.

tükrözés

A felületeken megkereshetjük és körberajzolhatjuk az összetartozó adatokat, amelyeket a szakterületi modellben osztályként reprezentálunk. Ezután minden egyes megjelenített, illetve beállítható adatelemhez határozzuk meg, hogy az az osztály mely attribútumára vonatkozik. A kapcsolatot az osztály és az attribútum nevéből felépített kifejezéssel adhatjuk meg, például `Tanfolyam.ár`, `Tanfolyam.véglegesítve`. A kifejezés vonatkozhat egy attribútum és egy érték egyenlőségére, például `Kép.elforgatás=180`.

A felületeknek az osztályra vonatkozó vezérléseit az attribútumok mintájára az osztály és a művelet nevéből képzett kifejezéssel adhatjuk meg, például `Készlet.rendelés()`.

A módszerrel kigyűjthetjük a szakterületnek a felhasználói felületeken megjelenő osztályait, valamint azok alapvető attribútumait és műveleteit is. A tükrözést a későbbiekben, a tervezés során valósítjuk meg.

A felhasználói felületeknek el kell érniük a megjelenített objektumokat. A *tükör* `(mirror)` osztályok valójában a szolgáltató `(service)` osztályok speciális változatai, amelyeken keresztül elérhetők és módosíthatók a kapcsolódó objektumok attribútumai, illetve meghívhatók bizonyos műveletei. A tükör osztályok technikáját használhatjuk az adatbázisban tárolt vagy a hálózat más pontján elhelyezkedő adatok interfészeként is.

tükör osztályok

Bizonyos esetekben a felhasználói felületeken keresztül közvetlenül nem érjük el a szakterületi modellünk objektumait. Ekkor újabb vezérlő `(control)` osztályokat, például szolgáltató `(service)`, esetleg tükör `(mirror)` osztályokat kell közbeiktatnunk.

Pontosítás megvalósítása

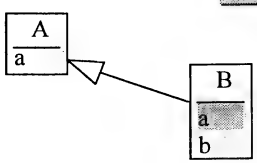
Az általánosítás/pontosítás elvét technikai szempontból öröklésként értelmezzük, az ős jellegzetességeinek átvételeként. Az öröklés több különböző módon is megvalósítható, de az OO környezetek csak a legegyszerűbb változatot valósítják meg automatikusan. A művelethez tartozó megfelelő működést a „kötés" választja ki. Az egyszerű pontosítás és az aggregáció a program kódjában azonos módon jelenik meg, így lehetőséges, hogy egy általánosítást aggregációként reprezentáljunk, vagy fordítva.

öröklés

A *pontosítás* megvalósításának több lehetséges módja is van. Az általánosítás-pontosítás az elv, melyet *öröklésként* `(inheritance)` értelmezünk, ami azt jelenti, hogy a pontosított változat rendelkezik, „örökli" az ősosztály jellegzetességeit is, melyeket ezután átdefiniálhat vagy kiegészíthet újabb jellegzetességekkel.

Az objektumorientált környezetek többé-kevésbé automatikusan megvalósítják az öröklést. Ha például bejelöljük, hogy egy `X` alosztály az `A` ősosztály pontosítása (`X :A`), akkor az OO környezet az `A` jellegzetességeit automatikusan „beemeli" az `X` osztályba, melyeket ezután (műveletek esetén) felüldefiniálhatunk vagy kiegészíthetünk. Adatok, azaz attribútumok esetén a „beemelés" mindössze úgy történik, hogy az ősosztály attribútumai az alosztály kiegészítő attribútumai előtt is megjelennek.

160. Ábra: Attribútumok átvétele



műveletek átvétele

A műveletek átvétele delegációval történik. Ha az `X` alosztálynak meghívjuk egy olyan műveletét, amely az `A` ősosztályra tartozik, akkor az alosztály egyszerűen az ős azonos műveletét hívja meg, azaz delegálja a hívást, míg egy átdefiniált vagy új művelet esetén már az alosztály művelete hajtódik végre. A többalakúság, a polimorfizmus egy kiegészítő technikát is igényel, mivel az ősosztályként tekintett objektum lehet egy olyan változat, amely bizonyos műveleteket átdefiniált, így a híváskor a *kötéssel* `(binding)` meg kell határozni, hogy adott művelethez mely metódus tartozik. A probléma leggyakoribb megoldása az objektumokhoz hozzárendelt (általában egy „nulladik" attribútumként felvett) *virtuális metódus tábla* (VMT), melybe az OO környezet felsorolja a műveletekhez tartozó módszerek címeit. A művelet átdefiniálásakor a táblázatban a műveletnek megfelelő helyre az új metódus címe kerül, így az ősosztályként tekintett változat esetén is a megfelelő metódus hajtódik végre.

kötés

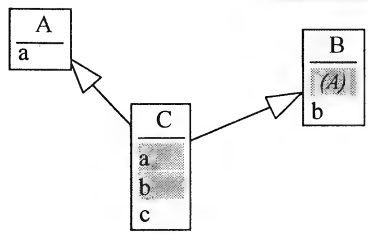
VMT

többszörös osztályozás megvalósítása

Minden változatot az ősosztályaként is el kell tudnunk érni. Ez a követelmény többszörös osztályozás *(multiple classification)* esetén megnehezíti az egyesített osztályok *(join class)* megvalósítását. Egy `A`, `B` és `C` ősosztályú `X` alosztály esetén (`X: A,B,C`) az attribútumokat úgy kell elrendezni, hogy az alosztály bármely ősosztályaként is tekinthető legyen. Általában az történik, hogy a `B` „elöl" az `A` attribútumainak megfelelő helyet kihagyja, a `C` pedig az `A` és `B` attribútumainak megfelelő helyet. Az ősosztályok így „összefésülhetők" és az egyesített osztály minden ősosztályaként is használható. Ez azonban azt igényli, hogy egy (például az `A`) kivételével a többi ősosztályt és az alosztályt egyszerre kell fordítani. Műveletek esetén a virtuális metódus táblát is azonos módon kell összefésülni, vagy az alosztályban az alosztálynak és minden ősének megfelelően külön táblákat kell elhelyezni és közülük az éppen megfelelőt felhasználni.

Bizonyos nyelvek egyszerűen megtiltják a többszörös öröklést. Például a Java csak egyetlen ősosztály megadását engedélyezi, ugyanakkor tetszőleges számú interfész is implementálható.

161. Ábra: Többszörös öröklés



késői kötés

A műveletekhez a megfelelő metódust a virtuális metódus tábla segítségével tudjuk hozzákötni, hozzárendelni. Ezt a módszer *késői kötésnek* *(late binding)* is nevezik, mivel a hozzárendelés az alkalmazás működése közben történik. Ha az alkalmazás működésének minden pontján egy műveletről pontosan tudjuk, hogy ahhoz mely metódus tartozik (például azért, mert nem polimorfikus, azt a változatok nem fogják felüldefiniálni), akkor ezt bizonyos nyelvekben bejelölhetjük és így hatékonyabbá tehetjük a kódot. Ekkor az OO környezet már fordítás során hozzárendeli a hívott művelethez a megfelelő módszert, így ezt a megoldást *korai kötésnek* *(early binding)* nevezzük. A késői kötés az általánosabb és biztonságosabb, a korai kötés viszont a hatékonyabb, de speciálisabb és csak nagyon ideális körülmények között használható megoldás. A korai kötés egy különösen veszélyes hibaforrás, mivel gyakran kiderül, hogy a fordítás során mégsem állapítható meg a megfelelő metódus, ezért csak nagyon indokolt esetben célszerű használni.

korai kötés

Az OO környezetek az öröklést automatikusan csak a legegyszerűbb változatban valósítják meg. Bonyolultabb helyzetekben, dinamikus osztályozás, többszörös típus, illetve többszörös öröklés esetén a hiányzó automatizmust „kézzel" kell pótolnunk.

asszociációként megvalósítás

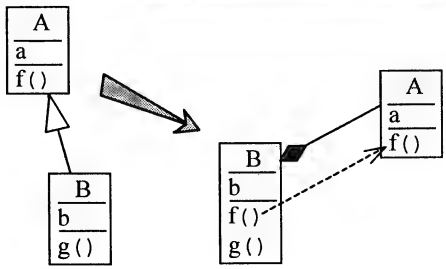
Az `A` ősosztály `X` alosztálya (`X :A`) esetén a pontosítás asszociáció (pontosabban kompozíció) felhasználásával is megvalósítható, mely a külön felvett alosztályból az ugyancsak külön felvett ősosztály felé vezet. Ekkor az alosztályban megismételjük

az ős műveleteit, az ősosztályra tartozókat egyszerűen delegáljuk, míg az átdefiniált vagy újonnan felvett műveleteket „helyben" végrehajtjuk. Többszörös öröklés esetén a műveletet az adott ősosztálynak megfelelő irányba kell delegálni.

A technika egy hátránya, hogy nem használható adatelemek, például attribútumok vagy asszociációk esetén. A probléma egy lehetséges megoldása, ha az adatelemeket a ***Q*** modellező módszerhez hasonlóan funkcionálisan értelmezzük, azaz műveleteken keresztül érjük el.

Az asszociációként történő megvalósítás dinamikus osztályozás, valamint többszörös változat és típus esetén is alkalmazható.

162. Ábra: Asszociációként történő megvalósítás



dinamikus típus

Érdemes még megemlíteni a *statikus* `(static typing)` és a *dinamikus típus* `(dynamic typing)` fogalmát. Statikus típus esetén az OO környezet már fordítás közben meg tudja állapítani, hogy egy objektumnak létezik-e egy adott jellegzetessége. Dinamikus nyelvek — mint például a `Smalltalk` — esetén lehetséges, hogy a típus dinamikus, azaz a jellegzetességek működés közben változhatnak, így csak fordítás közben dönthető el, hogy létezik-e egy adott jellegzetesség.

Pontosítás és kompozíció

Az egyszerű pontosítás és a kompozíció egy érdekessége, hogy a megvalósítás során („a kódban") ugyanúgy jelennek meg. Például egy `A` ősosztály `X` alosztálya esetén az alosztály adattagjainak elején az ősosztály adattagjai helyezkednek el. Egy `X` kompozit legelső `A` komponense esetén pontosan ugyanez a helyzet: az `X`-en belül az adattagok sorában előre az `A` adattagjai kerülnek, melyet az `X` további kiegészítő adattagjai követnek. Mindkét módszer közös elven, a másik osztály definícióinak „beemelésén" alapszik, amelyet általában *beágyazásnak* `(embedding)` neveznek.

beágyazás

163. Ábra: Pontosítás vagy kompozíció

modell ellenőrzése

A modellezés helyes módszere a valós világ fogalmai közötti tényleges viszony vizsgálatával határozható meg. Általánosítás-pontosítás viszonyt kell alkalmaznunk, ha az `X`-be beágyazott `A` osztály egy általánosabb fogalmat ír le, így az `X` minden objektuma egyben az `A` objektuma is. Ekkor az `A` az általánoshoz tatozó konkrét objektumok bővebb halmazát határozza meg. Ha az `A` az `X` része, akkor azt kompozícióként kell jelölni. Képletesen: kompozíció esetén a valós világban az `A` az `X`-en belül helyezkedik el, míg pontosítás esetén az általánosabb és a pontosabb változat egyaránt ugyanazt a területet foglalja el. Például egy `Személy` és egy `Biztosított` a valóságban ugyanazt a helyet foglalja el.

reprezentáció

A megvalósítás közös technikája, a beágyazás lehetővé teszi, hogy egy pontosítást kompozícióként modellezzünk vagy fordítva. Képletesen: a típusok részként történő reprezentációja úgy jelenik meg, mintha egy `Kétéltű jármű` esetén meg tudnánk határozni, hogy annak mely részei („mettől-meddig") alkotják a `Szárazföldi járműt` és melyek a `Vízi járműt`.

Az általánosítás „vagy" kapcsolatot jelent, mivel az absztraktként tekintett elem az „vagy" az egyik változat, „vagy" a másik változat, stb.

A többszörös pontosítás és a kompozíció egyaránt „és"-jellegű kapcsolat, mivel az egész az egyik részéből áll „és" a másik részéből áll..., vagy az egyesített osztály az egyik típusba tartozik „és" a másik típusba tartozik, stb.

Állapotdiagramok értelmezése

Az állapotdiagramokon állapotokat és közöttük vezető átmeneteket adunk meg, melyek általában külső események hatására következnek be. Az állapotok logikai értékű származtatott attribútumokként tekinthetők. A nem jelzés-jellegű és műveletekként megvalósított események esetén az állapotdiagramokat a megfelelő műveletek elő- és utófeltételeként értelmezhetjük, melyek általában egzaktul megadják az objektumnak a művelet hatására bekövetkező változását.

Állapotdiagramon általában a külső események hatására reakcióként bekövetkező állapotváltásokat ábrázolunk. Absztrakt jellegük miatt az állapotdiagramokat többféle módon is értelmezhetjük.

eseményként vagy őrszemként modellezés?

A modellezés során gyakran felvetődő kérdés, hogy az átmenetet eredményező külső helyzetet eseményként vagy feltételként, azaz őrszemként adjuk-e meg, mivel mindkettő egyaránt az objektum kiterjesztett állapotára vonatkozik. Például egy pattogó `Golyó` objektum esetén mi a különbség a `leért` esemény és a `[leért]` őrszem között? A ***Q*** módszer azt az egyszerű szabályt javasolja, hogy a környezetből érkező jelzést eseményként modellezzük, őrszemként pedig az objektum által végrehajtott, önmagára és környezetére, azaz a kiterjesztett állapotra vonatkozó tesztet jelöljük. Az elemzés kezdetén célszerű mindent eseményként modellezni, mivel ezzel a valósághoz közelebbi modellt kapunk. Ha az átmenet megfelelő irányát az objektum adott, egyetlen pillanatban végrehajtott teszttel is meg tudja határozni, akkor a feltételnek ezt a részét átírhatjuk őrszemmé, így egy hatékonyabb modellt kapunk, mivel az ellenőrzés művelete egyszerűbb, mint általában az esemény átvitele. A másik különbség, hogy az esemény félbeszakíthatja az állapotot, míg a teszt csak az átmenet során kerül ellenőrzésre.

Közbülső megoldást jelent az UML folyamatos tesztje, amelyet a `when` kulcsszóval adhatunk meg. A folyamatos teszt az állapoton belüli, elvileg állandó, a gyakorlatban azonban a végrehajtás bizonyos pontjain lefolytatott ellenőrzés, mely teljesülése átmenetet eredményezhet.

állapot értelmezése

A ***Q*** módszer minden elnevezett állapotot az objektum logikai értékű származtatott attribútumaként értelmez, mely igaz értéket ad, ha az objektum az állapotban van. Az őrszemek, illetve a folytonos tesztek feltételei ugyancsak a kiterjesztett állapotra vonatkozó logikai értékű kifejezések, de azokat nem mindig nevezzük el külön származtatott attribútumként. A módszer ajánlása szerint ezek a tesztek csak az absztrakt modellre, azaz a szakterületi fogalmakra hivatkozhatnak, implementációs részletekre azonban nem.

Az átmenetek és az események értelmezésekor el kell döntenünk, hogy az eseményeket olyan jelzéseknek tekintjük-e, melyek feldolgozás hiányában elvesznek. Ebben a megközelítésben, ha egy állapotban olyan jelzés érkezik, mellyel nincs címkézve átmenet, akkor az nem váltja ki az állapot megváltozását. Másik esetben a diagramot úgy értelmezhetjük, hogy az események csak bizonyos állapotokban vannak engedélyezve. Ha az állapotból nem vezet ki az eseménynek megfelelő címkéjű él, akkor annak egy hiba jelentését, azaz egy kivétel generálását kell eredményezni.

A módszer alapértelmezettnek a második esetet tekinti, míg a «signal» („jelzés") sztereotípiájú eseményt „opcionális" üzenetként értelmezi, amely feldolgozás hiányában elveszhet.

A nem «signal» üzenetek hatására bekövetkező állapotváltásokat ugyancsak többféleképpen implementálhatjuk. Az UML szerzői egy véges állapotú automataként történő megvalósítást ajánlanak. Azonban ez a megoldás elsősorban aktív objektumok esetén alkalmazható, holott állapotdiagramot többnyire passzív objektumra adunk meg. Aktív objektumok esetén az események vételét egyszerűen folyamatos teszttel is megvalósíthatjuk.

események értelmezése

Passzív objektumok az üzenet küldése előtti pillanatban a vezérléssel nem rendelkeznek, ezért az események vétele sem implementálható folyamatos tesztként. Az egyszerű üzeneteket általában a fogadó objektumban definiált, az üzenettel azonos nevű és paraméterű művelet hívásával valósítjuk meg.

Az objektumot elérő minden eseménynek egy-egy műveletet feleltetünk meg. Nem jelzés jellegű („nem opcionális") esemény esetén az állapotdiagramot úgy értelmezzük, hogy az esemény, illetve a művelet végrehajtása csak akkor engedélyezett, ha az objektum olyan állapotban van, amelyből vezet ki az eseménnyel címkézett él, különben hibát kell jelenteni. Az állapotdiagramot ezért az eseményeknek megfeleltetett műveletek előfeltételeiként értelmezzük, azaz a műveletet csak bizonyos állapotok esetén szabad végrehajtani. Az állapotdiagram ugyanakkor bizonyos utófeltételeket is jelent, mivel az események adott állapotból adott feltételek esetén az átmenetnek megfelelő másik állapotba (esetleg ugyanabba) válthatják át az objektumot.

A módszer tehát a normál, műveletként megvalósított események esetén az állapotdiagramot a következőképpen értelmezi: Gyűjtsük össze azokat az állapotokat, amelyekből vezet ki az eseménnyel megcímkézett él (például `S1` és `S2`), majd a műveletet az állapotok tesztjéből és az átmenetnek megfelelő újabb állapotok beállításából (például `S1_` és `S2_`) építsük fel, például:

```
eventX():
   if( isS1() )       {   setS1_();  }
   else if( isS2() ) {   setS2_();  }
   else throw new Exception("...");
```

Ha a híváskor az objektum egyetlen érvényes állapotban sincs, akkor egy kivétellel értesítjük a hívót. A szerkezet a feltételekkel (az állapotnak megfelelő elágazáson belüli tesztekkel), valamint az átmenethez kapcsolt akciókkal is értelemszerűen kiegészíthető.

VII.

Szakterületi elemzés

Az elemzés második szakaszában, a követelmények rögzítését követően a szakterület fogalmait, azok viszonyait és változásait ábrázoló szakterületi modellt állítjuk össze. A szakterületi modellt a követelmények alapján építjük fel, de az már nem az alkalmazás kívülről látható képét, hanem belső logikai szerkezetét ábrázolja, ezért alapvetően egy mérnöki, rendszerszervezői szempontot képvisel.

A szakterületi elemzést további két lépésre bonthatjuk. Az áttekintő szakterületi modell készítésekor adottnak tételezzük fel az objektumok struktúrájával, elérésével, felvitelével és törlésével kapcsolatos műveleteket. A részletes szakterületi modellben meghatározzuk a strukturális műveletek jellegét, valamint megadjuk a helyes működést leíró szabályokat és az azokat ellenőrző teszteket.

Diagramok a követelményelemzésben

A csomagokat és a használati eseteket ábrázoló diagramok mellett a követelmények rögzítésekor további diagramokat is alkalmazhatunk. A szakterületi elemek közötti jellemző helyzeteket objektumdiagramokkal szemléltethetjük. Az alkalmazási példákat aktivitásdiagramokkal, az objektumoknak a műveletsorozatok hatására bekövetkező változását pedig állapotdiagramokkal ábrázolhatjuk.

A ***Q*** modellező módszer az elemzés első fázisában, a követelmények rögzítése során alapvetően szöveges dokumentumokat ajánl. Az alkalmazás alapvető részeit ábrázoló csomagok és a funkcionális követelmények áttekintő térképeként szolgáló használati eset diagramok mellett a dokumentumokat további diagramokkal is kiegészíthetjük.

szemléltetés

A követelmények rögzítésekor az alkalmazás kívülről látható képét állítjuk össze. A dokumentumok alapvető célja a megrendelőkkel való egyeztetés, így a dokumentumoknak a felhasználók által értelmezhetőknek és ellenőrizhetőknek kell lenni. A követelmények rögzítése során ezért szemléltetésül csak olyan diagramokat alkalmazzunk, amelyek a felhasználók által még könnyen értelmezhetők, segítik a dokumentumok megértését és kizárólag az alkalmazás kívülről látható képére és működésére vonatkoznak.

A követelményelemzés során rögzített dokumentumokat egy előzetes felhasználói leírásban foglaltuk össze. A leírásban szemléltetésként további diagramokat is alkalmazhatunk.

objektumdiagram

A szakterület egyes *jellemző helyzeteit objektumdiagramokkal* ábrázolhatjuk. Ekkor az objektumokkal olyan szerepeket jelölünk, melyekbe több konkrét szereplő is behelyettesíthető. Ritkább esetben a diagram egy adott helyzetre is vonatkozhat. A diagramokat szemléletesebbé tehetjük, ha az objektumokat téglalapok helyett kis képekkel, ikonokkal ábrázoljuk.

aktivitásdiagram

Az *alkalmazási példákkal* bemutathatjuk, hogy az egyes használati esetek segítségével hogyan valósíthatók meg a felhasználói célok. Az alkalmazási példákban több aktor is szerepel, így azokat grafikusan *aktivitás-diagramokkal* ábrázolhatjuk. Az egyes tevékenységek (aktivitások) a használati eseteket, illetve a külvilág történéseit jelölik. Aktivitás-diagramokkal a használati esetek alkalmazásának időbeli sorrendiségét is rögzíthetjük.

állapotdiagram

A használati esetek általában egy vagy több objektum *változásával* kapcsolatosak. A változást *állapotdiagrammal* adjuk meg, ahol az átmenetek eseményei a használati eseteknek feleltethetők meg

Szakterületi elemzés

Az elemzés második részének célja a szakterületi modell kialakítása, amely a szakterület fogalmait, a közöttük lévő viszonyokat és azok változásait ábrázolja. A szakterületi elemzést további két lépésre bonthatjuk. Az áttekintő szakterületi modellben adottnak tételezzük fel az objektumok struktúrájával (elérésével, felvitelével és törlésével) kapcsolatos műveleteket. A részletes modell a strukturális műveletek jellege mellett az elemekre vonatkozó szabályokat és teszteket is tartalmazza.

A ***Q*** modellező módszer két részre bontja az alkalmazásfejlesztés első fázisát, az elemzést. A *követelményelemzés* az elkészítendő alkalmazással szemben támasztott igényeket gyűjti össze. A követelményeket a rendszer kívülről látható képének megrajzolásával határozzuk meg, amely a felhasználói célokat, a nagyobb egységeket (csomagokat), az elérhető funkciókat és a felhasználói felületeket is magában foglalja. A követelményeket alapvetően szöveges dokumentumokban rögzítjük, melyek a megrendelő által is áttekinthetők és ellenőrizhetők. A dokumentumokat egy adott alapszerkezetű előzetes felhasználói kézikönyvbe foglaljuk össze. A követelmények elemzésével a célunk a rendszer határainak megrajzolása, annak meghatározása, hogy mi kerüljön be és mi maradjon ki az alkalmazásból, valamint a fejlesztés erőforrásigényének becslése.

alkalmazás alapszerkezete

Az elemzés második része, a *szakterületi elemzés* már nem a felhasználók oldaláról közelít. A rögzített igények alapján, az alkalmazás kívülről látható képét elemezve megrajzolja a belső alapszerkezetet. Felépíti a követelményekben, a felhasználói felületeken és a funkciókban tükröződő belső struktúrát, azaz a szakterület fogalmait és a közöttük lévő viszonyokat. A követelmények leíró jellegével ellentétben a szakterületi elemzés már a megvalósítás tervezői, mérnöki szempontját képviseli.

A ***Q*** módszer a szakterületi modell elkészítését további két lépésre bontja. Az *áttekintő szakterületi elemzés* során adottnak tételezzük fel az adatok struktúrájával kapcsolatos műveleteket, azok elérését, felépítését és lebontását. A részletes szakterületi modell már a strukturális műveleteket is tartalmazza, de azokat csak vázlatosan, jellegként határozza meg. Például egy asszociáció szerepeinél az áttekintő modellben csak a számosságot adjuk meg, míg több számosság esetén a részletes modellben a kollekció jellegét is (pl.: `OrderedSet<>`, stb.) jelöljük.

szakterület fogalmai

Fontos, hogy az elemzés során, a követelmények rögzítése és a szakterületi modell összeállítása közben kerüljük a megvalósítással kapcsolatos technikai részleteket és informatikai fogalmakat. Az elemzés kizárólag a felhasználó és a szakterület fogalmait használhatja, így az adatokat és a működést is csak a felhasználó által még látható szintig követheti nyomon.

A szakterületi modellnek nem célja a felhasználóval és a megrendelővel történő egyeztetés, mivel már nem az igényeket, hanem a fogalmakat és a közöttük lévő viszonyokat rögzíti. Ugyanakkor egy műszaki végzettségű felhasználó értelmezheti és véleményezheti azokat.

A szakterületi modell készítésének kiindulópontjai a követelményanalízis során meghatározott használati esetek, valamint a felhasználói felületek, mely utóbbiakat speciális osztályoknak tekintjük.

Áttekintő szakterületi modell

A szakterületi modell kiindulópontjai a követelményelemzés során összeállított dokumentumok. A követelményeket „kívülről-befelé" a következő hierarchiában határoztuk meg: a felhasználói célokat a használati eseteknek kell megvalósítani, a használati eseteket pedig a felhasználói felületek segítségével érhetjük el. A ***Q*** módszer a szakterületi modellt ezért elsősorban a felhasználói felületek alapján építi fel.

A modell készítésének a *vázlata* a következő:

osztályok és attribútumok

- A felhasználói felületeken az invariancia-elv alapján rajzoljuk körbe az összetartozó adatcsoportokat, „információs csomópontokat", keressük meg a megfelelő szakterületi fogalmat és azt modellezzük *osztályként*.
  - Gyűjtsük össze az entitások adatszolgáltató és értéktároló felelősségeit, és azokat modellezzük *attribútumokként*.
  - A megfelelő szakterületi fogalom alapján nevezzük el az attribútumot.
  - Határozzuk meg az attribútumok típusát. Az elemzés során célszerű koncepcionális típusokat használni (`Int`, `Bool`, `Real`, stb.), melyeket a tervezés során feleltetünk meg konkrét programnyelvi típusoknak.
  - Az alaptípuson belül határozzuk meg az attribútum felvehető értékeit, például értéktartományát.

érték-osztályok

- Keressünk összetartozó, de egyediséggel nem rendelkező értékcsoportokat, melyeket *érték-* (`«value»`) *osztályokként* modellezünk.

műveletek

- Keressük ki a felhasználói felületekről az objektumok vezérlésével kapcsolatos elemeket, melyeket a megfelelő osztály (vagy osztályként reprezentált csomag) *műveleteként* modellezünk. Az áttekintő szakterületi modell készítésekor adottnak tételezzük fel az objektumok struktúrájával, például elérésével, felvitelével és törlésével kapcsolatos műveleteket.
  - A működési leírások segítségével gondoljuk végig és vázoljuk fel, hogy a műveletet milyen lépésekkel tudjuk megvalósítani. A végrehajtást azonban kizárólag a szakterületi fogalmakig, a felhasz-

náló által is nyomon követhető szintig szabad részletezni, így nem használhatunk a megvalósítással kapcsolatos informatikai technikákat.

- A művelet felhasználhatja önmaga további műveleteit is. Ha azok közvetlen végrehajtását nem engedélyezzük, akkor azokat jelöljük védettként `(protected)`, esetleg sajátként `(private)`.
- A művelet leírásakor soha ne avatkozzunk be más objektum „belügyeibe", hanem a felelősséget osszuk meg az együttműködő objektumok között. Ha a leírás egy részlete túlságosan sokszor hivatkozik egy objektum különböző attribútumaira és műveleteire (túlságosan sok „pontot" használunk), akkor az általában arra utal, hogy a lépéseket célszerű a használt objektum egy új műveleteként felvenni.
- A felelősségek megosztását egyszerűbb esetben interakciós (szekvencia vagy együttműködési) diagramon ábrázolhatjuk. Bonyolultabb műveletet, például több objektumot érintő, esetleg párhuzamos, illetve szinkronizációt igénylő tevékenységet aktivitásdiagrammal szemléltethetünk.

- A felelősségek megosztása esetén ellenőrizzük, hogy a hívás helyén vagy egy megelőző ponton elérhetők-e az együttműködők. Ha nem, akkor az osztálynak strukturális módon, *asszociációval* kell biztosítani a kapcsolatot. *asszociáció*
  - Az asszociációk felvételekor jelöljük be a navigációs irányt.
  - Az asszociáció megadásakor azonnal jelöljük be a szerepek számosságait, hogy egy konkrét kapcsolatban hány objektum vehet részt.
  - Nevezzük el az asszociációt és a szerepeket, ha ezzel érthetőbbé válik a modellünk. A névnek nem szabad megegyezni a szerepeknél elhelyezkedő osztályok elnevezéseivel, mivel azt alapértelmezett névnek tekintjük.

- Ellenőrizzük és pontosítsuk az *elérés és kapcsolódás módját*. *kapcsolódás módja*
  - Alkalmazzunk minősítőt `(qualifier)`, ha a kapcsolatban egy érték alapján szeretnénk egy vagy több objektumot kiválasztani.
  - Ellenőrizzük a szerepek osztályainak attribútumait, hogy azok nem-e asszociációs attribútumok. A próbát úgy végezhetjük el, hogy a szerepek multiplicitásait a legáltalánosabb „többre" (*) változtatjuk, majd ellenőrizzük, hogy egyértelműek maradtak-e az attribútumok. Az asszociációs osztályokat a tervezés során bizonyos esetekben „beintegrálhatjuk" az egyik szerep osztályába.
  - Vizsgáljuk meg, hogy a kapcsolatnak van-e kihatása az objektum élettartamára és attól függően jelöljük az aggregációt vagy a kompozíciót.

általánosítás-pontosítás viszonyok

- Határozzuk meg az osztályok közötti *általánosítás-pontosítás viszonyokat.*
  - Ha a felületek bizonyos helyein az általános fogalom, a jelleg jelenik meg, más helyeken pedig annak pontosított változatai, akkor jelöljük a fogalmak közötti viszonyt.
  - Ha bizonyos osztályok közös jellegzetességekkel rendelkeznek és a reprezentált fogalmaknak az általánosabb szakterületi fogalma is létezik, akkor a közös jellegzetességeket emeljük ki egy ősosztályba.
  - Jelöljük, ha az osztály absztrakt, azaz közvetlenül nem készíthetők példányai.
  - Használjunk megkülönböztetőt, ha az ősosztályként tekintett objektumról meg szeretnénk állapítani, hogy az mely alváltozatba tartozik.
  - Jelöljük, ha bizonyos változatok kizáróak.

kapcsolatok szűkítése

- Vizsgáljuk meg, hogy az *objektumok közötti kapcsolatok* nem szűkíthetők-e tovább. Asszociációk esetén a szerepnél jelölhetjük, ha valamely ősosztályra vagy interfészre egyszerűsítjük a kapcsolatot.

csomagokra bontás

- A szorosabban összetartozó osztályokat bontsuk *csomagokra.* A felbontásnak két iránya lehetséges:
  - *Rétegre* `(layer)` bontás esetén a magasabb szintű csomagok az alacsonyabb szintűekre épülnek.
  - *Partíciókra* `(partitions)` bontás esetén azonos rétegben a részben független, más funkciókört, feladatcsoportot ellátó csomagokat határozhatunk meg.
  - Jelöljük a csomagok közötti függőségeket.

A felhasználó által közvetlenül elérhető használati eseteket csoportosító csomagok az alkalmazásban futtatható programokként jelennek majd meg. Ezeket a csomagokat célszerű egy teljesen különálló felső rétegben megadni.

felső szintű csomagok

A felső szintű csomagok feladata általában arra „korlátozódik", hogy a felhasználói felületek működtetésével tegyék lehetővé a használati eseteik tevékenységének végrehajtását a szakterületi modell megfelelő elemein. A felhasználói felületek ezért a futtatható programoknak az állapotai, melyeket osztályként reprezentálunk. Diagramon ezt úgy ábrázolhatjuk, ha a felső szintű csomagokon belül, grafikusan megjelenített lényegi tartalomként felvesszük a felhasználói felületeket és azokat a függőségekként reprezentált átmenetekkel, a vezérlésekkel kapcsoljuk össze. Az átmeneteket általában a felhasználói vezérlésekkel címkézzük meg (pl.: `Ok`), de ahhoz feltételek és akciók is megadhatók. A felületeket csak a nevükkel vagy teljes, konkrét megjelenési formájukban is ábrázolhatjuk. Mivel a felületek bizonyos, például beállításokat végrehajtó elemei magára a futtatható programra vonatkoznak, ezért a csomagon belül felvett, a csomaggal azonos nevű, de nagybetűvel kezdődő osztállyal reprezentáljuk a program adatait.

Az elemzés során a felhasználói felületek objektumokra vonatkozó elemeit a szakterületi modell elemeivel még csak szöveges módon, egy-egy kifejezés megadásával kapcsoljuk össze. Diagramon ezt legfeljebb úgy adjuk meg, hogy a megjelenési formájukban ábrázolt felhasználói felület megfelelő eleméhez kapcsoljuk a szöveges megjegyzésként megadott kifejezést vagy az elemtől egy függőséget húzunk a megfelelő szakterületi osztályhoz (vagy annak jellegzetességéhez), melyet a kifejezés további részével címkézünk meg.

Részletes szakterületi modell

strukturális műveletek

Az áttekintő szakterületi modellben az objektumok struktúrájával kapcsolatos műveleteket adottnak tételezzük fel. A részletes modell készítésekor a modellünket a strukturális műveletekkel is kiegészítjük, de azok leírását csak a jellegekre korlátozzuk. Több számosságú szerep esetén például bejelölhetjük, hogy a kollekciót rendezett halmazként (`OrderedSet<>`) fogjuk tárolni, de a tényleges tárolás módját és a műveletek végrehajtását csak a későbbi tervezés során adjuk meg.

A részletes modell elkészítése általában a következő kiegészítéseket jelenti:

- Kapcsolatok esetén határozzuk meg a *változtathatóságot*, hogy a kapcsolatok módosíthatók, felvihetők vagy törölhetők.
- Határozzuk meg a *kapcsolatok*, illetve több számosságú szerepek esetén a kollekciók *jellegét*. A jelleget generikus osztályok mintájára adhatjuk meg, például `List<Alkalmazott>` (kiejtve: *„list of …"*). A publikálás technikáját felhasználva a jelleg műveletei az ellentétes oldalú osztályban is megjelennek.
- Adjuk meg *konstruktorokként* az új objektum felvitelének lehetőségeit és *destruktorként* a törléskor végrehajtandó műveleteket.
- Jelöljük be, hogy mely osztályok objektumait, illetve azok mely attribútumait szükséges tárolnunk. A szakterületi modell osztályait általában mind tároljuk, illetve a futtatható programok egyes beállításait is esetenként meg szeretnénk őrizni. A *tárolás* megadására az UML a `{persistent}` megszorítást ajánlja, illetve `{transient}` megszorítással hangsúlyozhatjuk, ha az osztályt vagy az attribútumot nem kívánjuk tárolni. A tárolás módját (adatbázis, adatfájl, `.ini`-fájl, stb.) és annak műveleteit a későbbiekben, a tervezés során határozzuk meg.
- Keressük meg és jelöljük be a szükséges *konverziókat*. A konverzió során egy objektum információtartalma alapján egy másik objektumot állítunk elő (`toX`), vagy az objektumot egy másik típusként tekintjük (`asX`).
- Határozzuk meg és jelöljük be az üzenetek, illetve a generált események jellegét. A módszer a jelleg megadására ugyancsak generikus szerkezetet ajánl, például `Signal<OutOfOrder(x)>`, `Exception<FileNotFound>`. A jelleggel utalhatunk az üzenet küldésének és fogadásának módjára.

szabályok

A strukturális műveletek mellett a modellünket ki kell egészíteni az elemekre vonatkozó szabályokkal, megszorításokkal:

- Osztályok, attribútumok és kapcsolatok esetén adjuk meg a felvehető értékekre vonatkozó *invariánsokat*. Invariánsokkal a felhasználói felületeken megadható értékeket is korlátozhatjuk.
- Műveletek esetén keressünk *elő- és utófeltételeket*.
- Kiegészítésként keressünk az egymás után végrehajtott műveletekre vonatkozó szabályokat. Ezek a különleges korlátozók, melyeket *algebrai szabályoknak* nevezhetünk, nem a működés bizonyos pontjain szűkítik a felvehető értékek tartományát, hanem a dinamikára és a transzformációkra vonatkoznak, a műveletek közötti összefüggéseket, az egymás utáni végrehajtás szabályait írják le. Például a helyes működésű verem `(stack)` osztály esetén a `push(x)` műveletet követő `pop()` értéke pontosan `x`. A módszer az állapotdiagramokat is a működésre vonatkozó algebrai szabályokként értelmezi.
- A követelményelemzés során elkészített, a használati esetek együttes működését bemutató *alkalmazási példákhoz* hasonlóan állítsunk össze műveletsorozatokat és adjuk meg a végrehajtás várható eredményére vonatkozó szabályokat, amelyek az alkalmazás helyes működését szemléltetik.

Milyen részletesek legyenek a szabályok?

szabályok részletezése

Le kell írniuk, hogy a felhasználói felületeken milyen értékeket, illetve értékkombinációkat adhatunk meg. Az ellenőrzést természetesen az alkalmazásba is be kell építenünk. Például egy `X` osztály `a: String` attribútumába betöltendő egész érték ellenőrzését megadhatjuk az `X` egy `checkA(a:String): String` műveletével. Nem megfelelő érték esetén a művelet egy szöveges kivétellel üzenhet, melyet a felhasználói felületen is célszerű megjeleníteni („`Adja meg a tanfolyamtípus nevét!`"), például egy állapotsorban vagy egy dialógusablakban. Az ellenőrző művelet a szöveges érték átalakítását is elvégezheti, például az értéket a megfelelő típusúvá konvertálhat. A technikát alkalmazva az attribútumérték beállítása a következőképpen történik: `setA(checkA(a_))`. Több érték együttes ellenőrzését a teljes osztályra vonatkozó invariánssal ellenőrizhetjük. Az invariáns megadható egy `check()` művelettel, ami az esetleges inkonzisztenciáról kivétellel értesíti a hívót.

Műveletek esetén célszerű előfeltételként megadni, hogy a művelet mely értékekre működik helyesen. Az utófeltétellel olyan szabályt határozhatunk meg, amelynek a helyes működést követően teljesülni kell. Egyszerűbb műveletet az elő- és utófeltételek egyértelműen is meghatározhatnak, ebben az esetben tehát a művelet azonos az utófeltétellel, így azt nem szükséges külön ellenőrizni. Ha a művelet több helyről is hívható, akkor az előfeltétel ellenőrzését célszerű be is építeni a programba. A paraméterek helyességét a művelet elején, annak egy „bevezető" szakaszában ellenőrizzük és a szabály sérüléséről kivétellel értesítsük a hívókat. Például egy verem `pop` műveletét a következő bevezető utasítással kezdhetjük:

```
if( isEmpty() ) throw EmptyStack(this);
```

Az utófeltételeket a műveletek végén hasonló módon vehetjük fel.

Az elemzőnek lehetőséget kell adni arra, hogy a már megtervezett alkalmazást megvalósító programozó folyamatosan ellenőrizni tudja önmagát, az implementáció végén pedig be tudja bizonyítani, hogy valóban a rögzített modellt készítette el. Az elemzőnek ehhez az invariánsok, elő- és utófeltételek, az algebrai szabályok és az alkalmazási példák alapján össze kell állítani példaadatokat, azokon végrehajtott műveletsorozatokat és az eredményre vonatkozó szabályokat, melyekkel a helyes működés ellenőrizhető. A teszteknek automatikusan, külső beavatkozás nélkül kell végrehajtódni — szerencsére az objektumorientáltság technikái, a műveleteikkel önállóan, felhasználói felületek nélkül is vezérelhető objektumok erre lehetőséget adnak. A teszteket a speciális argumentummal indított alkalmazással, esetleg külön tesztalkalmazással futtathatjuk le. A hibás esetekről és annak körülményeiről itt is kivétellel tájékoztathatjuk a hívót. A teszteket az alkalmazás egészére és külön a csomagokra, esetleg annak kisebb egységeire, például az egyes osztályokra is meg kell adni. Az automatikus tesztekkel minden módosítást követően gyorsan és egyszerűen ellenőrizhetjük, hogy helyes maradt-e a működés.

Csomagok és osztályok

A szakterületi modell alapvetően strukturális szempontból írja le a szakterület fogalmait, a közöttük lévő viszonyokat és azok változásait. A követelmények áttekintését a használati eset diagramok segítségével tehetjük meg, valamint azok köré szervezhetjük az igényeket leíró többi dokumentumot. A mérnöki megközelítésű szakterületi modellben a főszereplők az osztályok, illetve az azokat csoportosító csomagok.

A csomagokat és az osztályokat a következő alapszerkezettel adjuk meg:

- Az osztály (vagy csomag) jellegét, áttekintését foglaljuk össze néhány sorban, mely a modellben használt rövidített elnevezésének az értelmezése. Az áttekintésben adjuk meg az osztály (vagy csomag) célját, más néven az osztállyal szemben támasztott követelményeket `(requirement)`. *követelmények*
- Soroljuk fel, hogy a célt milyen felelősségekkel `(responsibility)` valósítjuk meg. A célt legfeljebb három-négy felelősségre bontsuk fel, melyek az alacsonyszintű követelményeket jelentik. *felelősségek*
- Sorra véve a felelősségeket határozzuk meg, hogy azokat mely jellegzetességekkel, attribútumokkal, műveletekkel, illetve kapcsolatokkal valósítjuk meg. A jellegzetességeket ugyancsak lássuk el egy rövid leírással. *jellegzetességek*

A részletes szakterületi modell kidolgozását követően, a tervezés során a rendszert informatikai környezetbe ültetjük át. A tervezésnek olyan részletességű dokumentumot kell előállítani, amelyből a szakterülethez és a kiválasztott környezetekhez értő programozók már önállóan meg tudják valósítani és a tesztekkel le is tudják ellenőrizni az alkalmazást.

VIII.

Implementációs diagramok

Implementációs diagramokkal a modellünkben ábrázolt logikai szerkezet megvalósításával kapcsolatos döntéseket, például a fizikai elrendezést ábrázolhatjuk. Az UML kétfajta implementációs diagramot, a komponens- és az alkalmazási diagramot definiálja.

Komponens-diagramok segítségével az alkalmazás fizikai szoftver-alkotóelemeit, azok összefüggéseit és a közöttük lévő függőségeket ábrázoljuk.

Alkalmazási-diagramokkal az alkalmazással kapcsolatban álló, annak működését biztosító fizikai hardvert, számítógépeket, illetve egyéb egységeket és azok egymás közötti kapcsolatait szemléltetjük. Alkalmazási-diagrammal jól ábrázolható a több számítógépen futó, például kliens-szerver alkalmazások fizikai felépítése és az azokon működő komponensek.

Komponens-diagramok

Komponens-diagramok segítségével az alkalmazás fizikai építőelemeit és az azok közötti viszonyokat, például függőségeket ábrázoljuk. A diagramok alkalmasak a kész alkalmazás fizikai szerkezetének vázolására, valamint a fejlesztés során a használt fájlok közötti viszonyok szemléltetésére.

Az UML-ben definiált kétfajta implementációs diagrammal a megvalósítással kapcsolatos döntések ábrázolhatók. Komponens-diagramokon a komponensek, lényegi tartalmuk, valamint a kapcsolódási módjaik szerepelhetnek.

komponens

A komponensek a fejlesztés alatt álló, vagy a már kész alkalmazás fizikai alkotóelemeit jelölik. A rendszer moduljai is komponensek, melyek többnyire megfeleltethetők a csomagdiagram bizonyos csomagjainak.

Komponensek például a

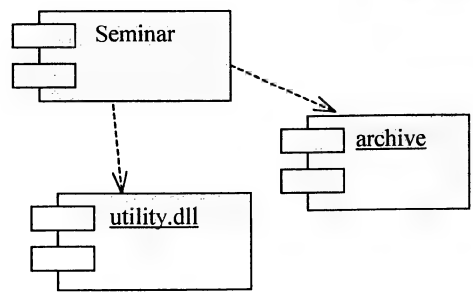
- forrás-állományok,
- kód-állományok `(object-file)`,
- programkönyvtárak,
- futtatható állományok,
- dokumentumok,
- adatfájlok, ...

A komponens egyszerűen valami olyasmi, ami például egy `copy` (vagy `cp`) paranccsal másolható, tehát többnyire egy állományként jelenik meg.

jelölés

A komponens jele egy téglalap, mely baloldalán két téglalap alakú címkét veszünk fel. A komponensek alakzatában lényegi tartalomként ábrázolhatók a fontosabb osztályok. A komponensnél sztereotípia (a név előtt) és megszorítás (az elnevezés után) is megadható.

164. Ábra: Komponensek



Komponens-típust a nevével azonosítunk, pl.: `Seminar`.

Konkrét komponenst az aláhúzással jelölt nevével és/vagy típusával adunk meg: `komponensNév :komponensTípus` vagy `archive`.

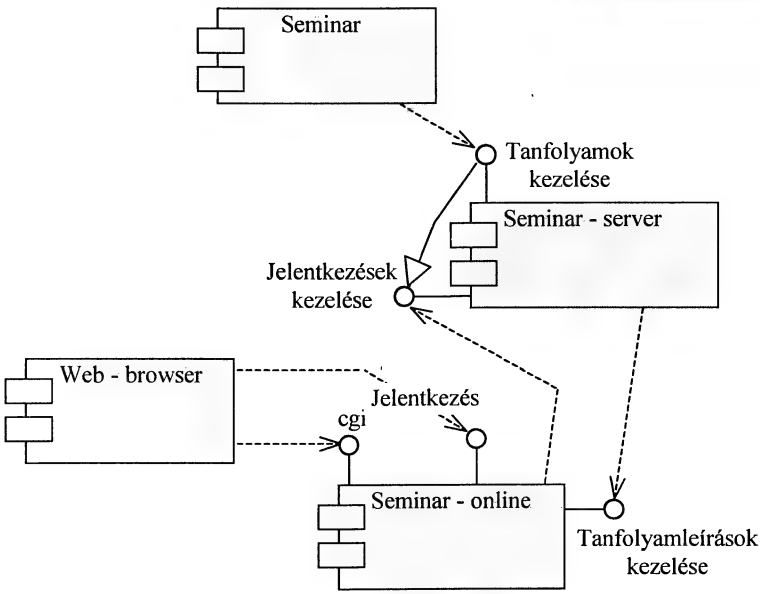
Komponensek esetén az UML öt sztereotípiát definiál:

- «executable» *(futtatható)* jelöli a futtatható programot tartalmazó állományt.
- «library» *(könyvtár)*: programkönyvtár, mely lehet statikus (pl. lib) vagy dinamikus (például dll) könyvtár is.
- «table» *(tábla)*: adatbázistábla.
- «file» *(fájl)*: forráskódot vagy adatot tartalmazó fájl.
- «document» *(dokumentum)* jelöli a dokumentumot tartalmazó állományt.

Újabb sztereotípiákat is felvehetünk, de az UML által előredefiniáltakat nem szabad eltérő értelemben használnunk.

A komponens ráépülhet más komponensekre, így felhasználhatja azok definícióit. A használatot függősségként ábrázoljuk. *függőség*

165. Ábra: Interfészek



A végrehajtható kódot, eljárásokat, függvényeket és programokat tartalmazó, például «executable» vagy «library» komponenseket a funkcióikon keresztül érhetjük el. A funkciókat interfészekbe csoportosíthatjuk és realizációval vagy a rövidebb „nyalóka" *(lollipop)* jelöléssel ábrázolhatjuk a komponens által megvalósított interfészeket. A jelöléssel kiemeljük azokat a lényegi követelményeket, „célokat", amelyeket a komponens teljesít.

Az interfészek egyben a komponenssel való kapcsolattartás eszközei is. Egy függőséget az interfész segítségével is megadhatunk, így a komponens használatát az interfészre korlátozhatjuk.

Alkalmazási-diagramok

Alkalmazási-diagramok segítségével az alkalmazás működtető hardver egységeit és az azok közötti kapcsolatokat ábrázoljuk. A diagramokkal jól ábrázolható az alkalmazás készülék-igénye, valamint a több számítógépen futó, például kliens-szerver alkalmazások fizikai felépítése és az azokon működő komponensek.

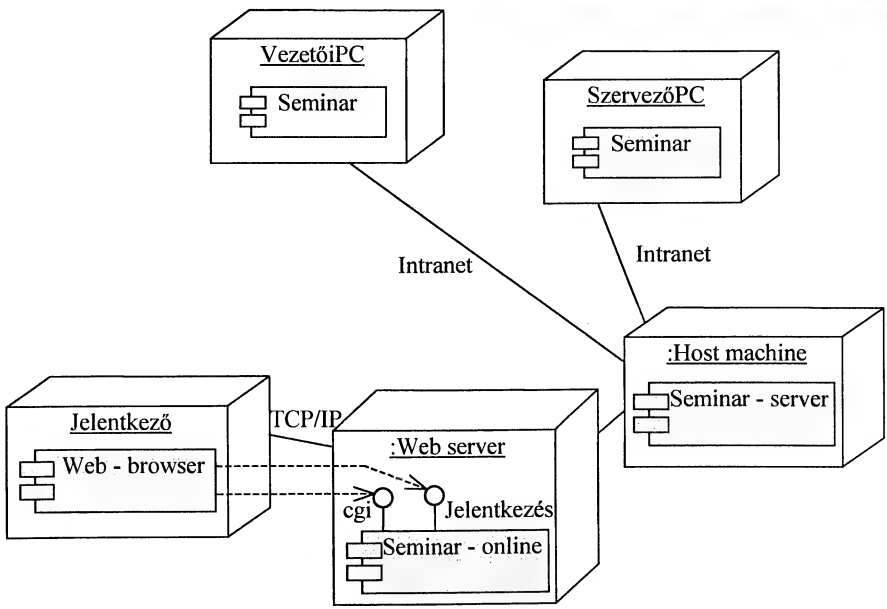
Az UML másik implementációs diagramja az *alkalmazási-diagram* `(deployment diagram)`, mellyel a rendszer hardverelemei és a közöttük lévő fizikai viszonyok ábrázolhatók. Az alkalmazási-diagramokkal a komponensek, illetve objektumok elhelyezkedését és továbbítását is szemléltethetjük.

csomópont

Az alkalmazási-diagram alapeleme a *csomópont* `(node)`, amelyet az UML a kocka alakzattal ábrázol. A csomópont `(node)` egy számítógépes egységet, a legtöbb esetben egy hardver elemet jelent.

A csomópontok lényegi tartalmaként ábrázolhatók a rajta elhelyezkedő komponensek, a lényeges adatelemek pedig objektumokként jeleníthetők meg.

166. Ábra: Alkalmazási-diagram



Csomópont-típust a nevével azonosítunk, pl.: `csomópontTípus`.

Konkrét csomópontot az aláhúzással jelölt nevével és/vagy típusával adunk meg: `csomópontNév: csomópontTípus`.

A csomópontnak sztereotípia és megszorítás is megadható. Az UML-nek csomópontok esetén nincsenek előredefiniált sztereotípiái, az UML szerzőinek példáiban azonban gyakran előfordul a következő kettő:

- «processor» *(processzor, feldolgozó egység)* jelöli a számítási kapacitással rendelkező egységet, például számítógépet, amelyen komponensek is elhelyezhetők.
- «device» *(készülék)*: számítási kapacitással nem rendelkező egység, például nyomtató, modem, fax, stb.

A csomópontok közötti asszociációkat *kapcsolatokként* (connection), azaz kommunikációs vonalakként értelmezzük. A kapcsolatok a kapcsolat típusával (pl.: TCP/IP) címkézhetők. A csomópontokon helyezkedhetnek el a komponensek.

kapcsolat

A csomópont-típusok osztály-jellegűek, így azoknak attribútumokat (például processzor típus, szükséges memória, stb.) és műveleteket (például készenlét, kikapcsolás) is definiálhatunk.

A csomópontok csomagokba is csoportosíthatók, valamint közöttük függőségeket, öröklést és asszociációt (például kompozíciót) is megadhatunk.

IX.

Tervezés és megvalósítás

A tervezés a szakterületi modellt informatikai közegbe ülteti át. A tervezés első lépésében, a stratégiai tervezés során az alkalmazás egészére vonatkozó döntéseket hozunk. Meghatározzuk az alkalmazási környezetet, amely az alkalmazás alapvető szoftverkomponenseit és a működést érintő hardverelemeket jelenti, valamint kiválasztjuk az implementációs platformot jelentő adatbázis-kezelőt, programozási nyelveket és technológiákat.

A tervezés második részében az alkalmazás részletes tervét állítjuk elő, amely a szakterületi logikát kiegészíti az azt megvalósító, működtető informatikai infrastruktúrával. Ebben a lépésben meghatározhatunk különböző összevonásokat és optimalizálásokat is.

A megvalósítás során először az alkalmazás önállónak tekintett részeit, a csomagokat dolgozzuk ki és ellenőrizzük. A kész elemet az alkalmazás egészébe integráljuk és az együttes teszttel ellenőrizzük a komponensek helyes kapcsolódási módjait.

Alkalmazási környezet

A két részre bontott tervezés első lépésében elkészítjük a stratégiai tervet. Ebben a fázisban a kifejlesztendő rendszerrel kapcsolatos átfogó, nagyobb léptékű döntéseket kell meghozni, amelyek meghatározzák az alkalmazás környezetét. Leírjuk az alkalmazás fizikai (szoftver, adat és hardver) elemeit, valamint a felhasznált adatbázis-kezelőt, programnyelveket és kiegészítő eszközöket.

A követelményeket rögzítő, majd a szakterületi modellt felépítő elemzés után a rendszerfejlesztés következő lépése a tervezés, amely a modellt informatikai környezetbe ülteti át. A tervezés célja olyan részletességű dokumentum összeállítása, mely alapján a kiválasztott környezetekhez és technológiákhoz értő programozók már önállóan implementálni és ellenőrizni is tudják az alkalmazást. Az elemzés elvi, koncepcionális megközelítésével ellentétben a tervezés alapvetően egy technológiai szemléletet képvisel.

A *Q* modellező módszer a tervezést ugyancsak két nagyobb részre bontja. A *stratégiai tervezés* feladata a megvalósítással kapcsolatos átfogó, nagyobb léptékű döntések meghozatala. A *részletes tervezés* a modell elemeire összpontosít, így az alkalmazást működtető infrastruktúrát dolgozza ki, valamint az egyszerűbb megvalósítás érdekében különböző összevonásokat hajt végre.

stratégiai tervezés

A stratégia tervezés az alkalmazás magas szintű felbontását és a használt eszközkészlet összeállítását jelenti. Mivel a döntések hiányában nem lehet alkalmazást fejleszteni, hasonló lépéseket szinte minden módszer tartalmaz, például ilyen az OMT módszerének *rendszertervezés* `(system design)` fázisa.

A stratégiai tervezés döntéseit a következő lépésekre bonthatjuk fel:

- Magas szintű, *futtatható* (`«executable»`) *komponensek*, „programok", „modulok" meghatározása. A felhasználóval való kapcsolattartás programjai általában azonosak a használati eset diagramok magas szintű csomagjaival. A szakterületi modell csomagjait nagyobb egységekbe olvasztjuk és azokat általában könyvtárakként (`«library»`) valósítjuk meg.
- Határozzuk meg az alkalmazással kapcsolatban álló *hardverelemeket* (számítógépeket és más készülékeket), azok kapcsolódási módját és a rajtuk elhelyezett komponenseket.
- Több számítógépen futó osztott `(distributed)` rendszer esetén alakítsuk ki a *magas szintű architektúrát*. Több felhasználós rendszer esetén gyakori például a kliens-szerver felépítés; ekkor a szakterületi modell elemeit általában a szerveren vagy szervereken tároljuk.
- Adjuk meg a komponensek *határfeltételeit*, az indításkor, leállításkor és a hiba esetén végrehajtandó tevékenységeket. Indításkor például bizonyos adatokról biztonsági másolatot készíthetünk, melyet hiba esetén automatikusan felhasználunk.

- Válasszuk ki a tárolandó adatok körét és határozzuk meg az *adattárolás jellegét* (pl.: szöveges fájl, adat-fájl, `.ini` fájl, stb.). Nagytömegű adat gyors elérése adatbázis-kezelőt igényelhet.
- Válasszuk ki a megvalósítás *környezetét*, amely a programnyelvek mellett saját vagy más által fejlesztett osztálykönyvtárakat és keretrendszereket *(framework)* is jelent.
- Készítsünk olyan *prototípusokat*, amelyek a kiválasztott adatbázis-kezelőnek és környezetnek az alkalmazás által igényelt felhasználási módját teszteli. Ez például relációs adatbázisban való tárolás és a Java nyelv kiválasztása esetén olyan rövidebb példaprogramokat jelent, amelyek az adatok elérésének és módosításának a módját és lehetőségeit tesztelik. A prototípusok készítése különösen az új technológiák, például külső fejlesztésű osztálykönyvtárak alkalmazása előtt lényeges.
- Határozzuk meg a komponensek közötti kommunikáció *protokollját*, valamint a globális erőforrások elérésének módját.
- Készítsünk *terhelési becsléseket* a szolgáltatás-jellegű komponensekre, valamint ezek alapján állítsunk össze a terhelést tesztelő terveket.
- Határozzuk meg az *optimalizálandó feltételeket*. Ezek többnyire az idővel, például az alkalmazás válaszidejével kapcsolatosak, de lehetnek a tárolt vagy a hálózaton (esetleg telefonvonalon) átvitt adatok méretére vonatkozó korlátozások is.
- Becsüljük meg az alkalmazás működtetéséhez *szükséges erőforrások* méretét. Az igényelt processzorsebességet, memóriaméretet nagymértékben befolyásolja a kiválasztott adatbázis-kezelő és környezet.

szemléltetés

A magas szintű komponenseket, azok függőségi viszonyait és elhelyezését *komponens*-, illetve *alkalmazási-diagramokon* ábrázolhatjuk.

alkalmazási környezet

A stratégiai tervezés csak lazán kapcsolódik a szakterületi modellhez, ezért annak alapvető lépéseit a követelmények rögzítése után is végrehajthatjuk. Az előzetes felhasználó kézikönyv a megrendelő igénye esetén ki is egészíthető az előzetes tervekkel, amelyeket például egy *Alkalmazási környezet* című részben adhatunk meg.

Részletes tervezés

A tervezés befejező részében előállítjuk az alkalmazás részletes tervét, amely a modellt a stratégiai tervezés során meghatározott informatikai és technológiai környezetbe ülteti át. A részletes tervezés során egyrészt felépítjük a szakterületi logikát megvalósító, működtető informatikai infrastruktúrát. Másrészt a részletes terv tartalmaz a fejlesztés felgyorsítása és esetleg a hatékonyabb működés érdekében tett különböző összevonásokat és optimalizálásokat.

A tervezés befejező részében, a részletes terv készítésekor az alkalmazás részleteire összpontosítunk, mely során a szakterületi modellt a stratégiai tervben meghatározott alkalmazási környezetre építjük rá.

informatikai környezet

A szakterületi modellben felépített logikát informatikai környezetbe kell átültetnünk, azaz a modell minden elemének egy-egy informatikai technikát kell megfeleltetni. A szakterületi modell elemei egy absztrakt megoldási módot, azaz több lehetséges megoldási változatot határoznak meg. A tervezés során ezek közül ki kell választanunk az elem használati módjának megfelelő technikát.

A megfelelően elkészített szakterületi modell a szükségesnél általánosabb belső szerkezetet, architektúrát ír le. A tervezés során dönthetünk úgy, hogy bizonyos összevonásokkal, származtatott attribútumok és kapcsolatok tárolásával, valamint egyéb optimalizálási technikákkal hatékonyabbá tegyük a rendszert. Az általános szerkezet megtartása a modell későbbi módosításai során térülhet meg, mivel a feladat kiterjesztései, a bővebb követelmények kevesebb egyszerűsítést engednek meg.

optimalizálás

összevonás

Az összevonásokat esetenként a használt technológia is megkövetelheti. Adatbázisban történő tárolás esetén gyakori a leszármazott osztályok jellegzetességeinek beintegrálása az ősosztályba, azaz a származtatást állapotként modellezzük. Az összevonás másik célja, hogy segítségével felgyorsítsuk a fejlesztést, így az aktuális változat elkészítésének ideje és a későbbi módosítási igények becslése alapján kell döntenünk.

hatékonyság

Az összevonásokat és programmódosításokat esetenként azért tesszük meg, hogy hatékonyabb rendszert valósítsunk meg. A hatékonyság sokkal kisebb probléma, mint amennyi figyelmet arra általában fordítanak. A nem hatékony rendszer leggyakrabban valamely alapvető tervezési hiba vagy a nem megfelelő technológia kiválasztásának az eredménye, nem pedig az általánosabb programszerkezetek hibája. Az „optimálisabb" kód eltér a feladat természetes megfogalmazásától, így a megvalósítása, értelmezése, áttekintése és későbbi módosítása is nagyságrendekkel nagyobb energiát és ezért nagyobb költséget is igényel. Célszerű először egy áttekinthető módon felépített működő (!) alkalmazást készíteni, melyet csak akkor optimalizálunk, ha a hatékonysági probléma a gyakorlatban bizonyítottan létezik. Ha a problémát nem alapvető szervezési hiba, vagy a nem megfelelő

technológia okozza, akkor az optimalizálandó helyeket a megfelelő mérő *(profiler)* programmal keressük meg. Gyakran a legváratlanabb helyeken tölt el sok időt a program, nem pedig ott, ahol mi gondolnánk.

infrastruktúra tervezése

Az infrastruktúra felépítését a következő lépésekben határozhatjuk meg:

- Az *osztályok* tervezésekor meghatározzuk az objektumok reprezentációját. A programnyelv osztály *(class)* fogalma mellett a tárolt *(persistent)* objektumok adatbázistáblákban is elhelyezkedhetnek.
- Adjuk meg a *kapcsolatok* megvalósításának módját. Adatbázistáblákban a kapcsolatok például idegen kulcsokként *(foreign key)* jelennek meg. Kliens-szerver alkalmazások esetén bizonyos objektumhoz kapcsolódó más objektumokat nem feltétlenül szeretnénk közvetlenül elérni, hanem csak tényleges felhasználás esetén. Ez azt jelenti, hogy a kapcsolat bizonyos szempontból tárolt származtatott kapcsolatként jelenik meg, melyet nem feltétlenül az objektumazonosító alapján építünk fel.
- A *kollekciókat* a szakterületi modellben jellegként (pl. `OrderedSet<Jelentkezés>`, azaz *„ordered set of ...”*) határoztuk meg. A tervezés során az egyes jellegekhez konkrét *tároló osztályokat* *(container class)* rendelünk. Célszerű az egyes jellegeket egy-egy könnyen ellenőrizhető, általános felépítésű tároló osztályként megvalósítani (halmazt például egy dinamikus, változó méretű tömbként — a Java nyelvben ez a `Vector` osztály). Az általános tároló osztályt az alkalmazás finomhangolásakor egy hatékonyabb változatra cserélhetjük, de kizárólag csak akkor tegyük ezt meg, ha a már kipróbált rendszerben hatékonysági problémák merültek fel és annak bizonyíthatóan ez a fajta tárolás az egyik oka.
- A nem függvényhívásként megvalósított *üzeneteket* a szakterületi modellben jellegként (pl.: `Signal<OutOfOrder>`) határoztuk meg. A tervezés során a jellegekhez konkrét megvalósítási módokat kell rendelnünk.
- A tervezés újabb osztályok felvételét, esetenként pedig a korábbi osztályok módosítását igényli. A tervezés során ezért célszerű ellenőrizni a rendszer *csomagokra* bontását, illetve az új osztályokat új csomagokba csoportosítani.
- Meg kell határozni a más objektumokat helyettesítő tükör *(mirror)* osztályok esetén a *tükrözés módját*. Ez a felhasználói felületek összekapcsolását is jelenti a modellelemekkel, de tükrözéssel megoldhatók még az objektumok átjuttatása a hálózaton, vagy az adatbázisban történő tárolás.
- Tervezzük meg az *algoritmusokat*. A szakterületi modell a bonyolultabb algoritmusoknak csak a jellegét (például: „rendezés”) határozta meg. A tervezés során kiválasztjuk a jelleg egy megvalósítási módját (pl.: „gyorsrendezés”). Ez a lépés a tesztek megtervezését is jelenti.

A szakterületi modell informatikai megvalósítása során végrehajthatunk különböző egyszerűsítéseket és optimalizálásokat. Az egyszerűsítéseket két fő szempont alapján célszerű meghatározni: vagy a fejlesztés felgyorsítása, vagy az alkalmazott technológia (például relációs adatbázisban történő tárolás) gyengeségeinek és hiányosságainak kiküszöbölése érdekében. Ha csak lehetséges, az optimalizálás vizsgálatát a már működő alkalmazás finomhangolásának idejére halasszuk. A gyakorlati tapasztalatok alapján programozás közben általában olyan műveleteket írunk át hatékonyabbá, amelyek végrehajtási ideje a teljes működés szempontjából nem is szignifikáns.

A követelményeket alul múló, nem hatékony működés esetén a nyilvánvaló optimalizálások (pl. hosszabb időtartamú számítások tárolása) mellett először a szervezést, az alkalmazás belső architektúráját célszerű ellenőrizni. A rendszer optimalizálandó helyeit az objektív szempontból ítélni tudó tesztprogramok `(profiler)` felhasználásával keressük meg.

összevonások

A részletes tervezés során például a következő összevonásokat és optimalizálásokat tehetjük meg:

- Hosszabb időtartamú számítások eredményeként előálló származtatott elemeket (származtatott attribútumokat, asszociációkat, osztályokat) *gyorsítótárakban* `(cache)` tárolhatjuk.
- *Kompozitba* esetenként beemelhetők az 1 számosságú komponensek.
- Bizonyos 1-1 vagy 1-opcionális számosságú *osztályok összevonhatók* egyetlen osztállyá.
- *Asszociációs osztály* esetén, ha egy kivételével az asszociáció összes szerepe legfeljebb 1 számosságú, a jellegzetességek beemelhetők a magasabb számosságú osztályba.
- Egy *ősosztály* (például `Személy`) *alosztályai* (pl. `Férfi`, `Nő`) esetenként az ősosztályba integrálhatók. Ekkor az ősosztályba az összes leszármazott osztály jellegzetességeit is beemeljük, azaz a változatot állapotként modellezzük.
- Az optimalizálás érdekében esetenként módosíthatunk a *műveletek végrehajtási sorrendjén*, vagy egyes műveleteket hatékonyabb algoritmusokra cserélhetünk.

Az alkalmazás részletes tervének olyan pontosan kell leírnia a feladatot, hogy azt a kiválasztott környezethez és technológiákhoz értő programozók önállóan megvalósítani és ellenőrizni is tudják.

informatikai infrastruktúra

A tervezés feladata, hogy az elemzés során meghatározott szakterületi logika alá egy működtető, informatikai infrastruktúrát „csúsztasson be". Az informatikai infrastruktúra valójában megoldási módok, „trükkök", technikák gyűjteményét és azok alkalmazását jelentik, mely során a szakterületi modellt átképezzük `(mapping)` az informatika világába.

osztálykönyvtárak

A kidolgozás a töredékére (esetenként az eredeti időtartam ötödére, néha a tizedére (!)) csökkenthető, ha rendelkezésre állnak a megfelelő osztálykönyvtárak és keretrendszerek `(framework)`. A Java nyelvet részben az teszi kiemelkedően produktív környezetté, hogy igen nagy mennyiségű osztálykönyvtárral rendelkezik, így az alkalmazást nem a „semmiből" kell megteremteni.

Megvalósítás

*A **Q** módszer a megvalósítást két tevékenységre bontja szét. Az implementáció és teszt során az alkalmazás adott részének, csomagjának kidolgozására és önálló tesztelésére összpontosítunk. A rész kidolgozását követően azt az alkalmazás egészébe integráljuk. Az integráció tesztjei az együttes működésre vonatkoznak.*

A **Q** módszer a részletes terv *megvalósításának* `(elaboration)` fázisát két elemre bontja. Az *implementáció és tesztelés* során a feladat részleteinek kidolgozására összpontosítunk. A feladatot először különálló részekként, azaz csomagokként tekintjük, így a figyelmünket nem kell megosztanunk több, kidolgozás alatt álló elem között. A csomagra megadott tesztek a csomag elemeinek helyes működését vizsgálják.

implementáció és teszt

A rész kidolgozását követően azt integráljuk, azaz összekapcsoljuk az alkalmazás egészével. Az *integráció* tesztjei a helyes kapcsolódási módra, az együttes működésre vonatkoznak. A teljes iterációra vonatkozó integrációs lépések már kész programokat `(release)` eredményeznek, amely a telepítés `(setup)` megoldását is igényli.

integráció és együttes teszt

A megvalósítás legnagyobb kihívása, hogy valóban a részletes tervben megadott rendszert dolgozzuk ki, mégpedig olyan módon, hogy az más által is áttekinthető és ellenőrizhető legyen. Az áttekintés lehetőségét egy egyszerű „ökölszabály" betartásával segíthetjük, mely szerint a program minden sorához legalább ugyanannyi megjegyzéssort is írjunk — az osztályok, illetve a jellegzetességek alkalmazásban betöltött szerepét áttekintő „értelmezések" mellett. A helyesség biztosításának egy közismert és igen hatékony módja a „kódolvasás", amikor a kidolgozó jelenlétében egy másik programozó értelmezi a program utasításait.

„Le roi est mort, vive le roi!"

A.

„Objektumokon túl"

Az objektumorientáltság történetében észrevehető egy határozott irány, amely szemlélete a klasszikus megközelítéssel ellentétben már nem a bezárt és önállóságukban létező objektumokra épül. A történet során fokozatosan előtérbe kerültek az objektumok külső kapcsolódási módjai, használati lehetőségei. Ugyancsak ebbe a trendbe sorolhatók az „objektumokon túli" szemléletet képviselő legfrissebb kísérleti tervezési és programozási módszerek.

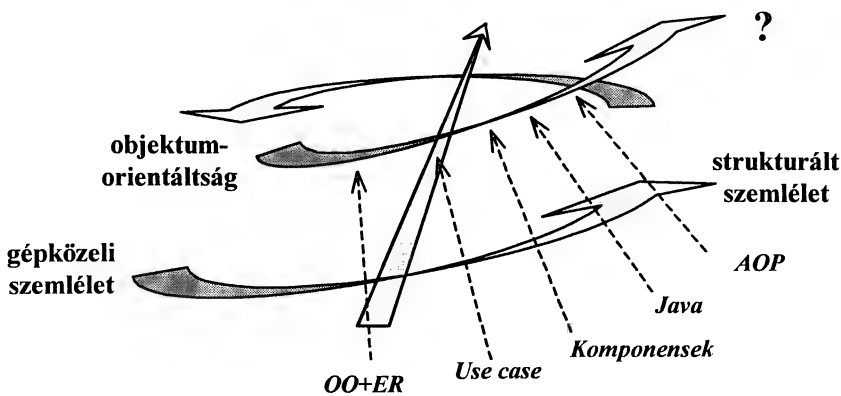
„Beyond objects"

Az informatikai paradigmák történetét vizsgálva becsléseket mondhatunk az elkövetkező években várható fejlődés alapvető irányairól. A „vegytiszta" OO megközelítés zárt objektum-fogalma helyébe az objektum használati és kapcsolódási módjai, az objektum külső vonatkozásai kerülnek előtérbe.

A *Q* modellező módszer egy érdekessége, hogy nem tisztán objektumorientált szemléletű, hanem úgy használja fel az OO technikákat, ahogy az a strukturált megközelítés eredményeit. Ezen új szemlélet irányába történő lassú elmozdulás, a „hagyományos" OO technikáktól történő elszakadás az informatika több területén is megfigyelhető. Ezért elképzelhető, hogy a fejlődés következő mérföldköve, a jövő paradigmája is majd hasonló jellegű koncepciókat alkalmaz.

Az informatikai szemléletek fejlődésében egy határozott irányvonal fedezhető fel. A jelölésrendszerek egyre inkább elszakadnak a gépi szemlélettől, és mindinkább a valósághoz közelebbi, *„természetesebb" ábrázolási módok* kerülnek előtérbe.

167. Ábra: Informatikai paradigmák fejlődése



spirál-jellegű fejlődés

A fejlődésben ugyanakkor egy *tézis-antitézis-szintézis* emelkedő *spirál-jellege* is megfigyelhető. A gépközeli programozás a feladatot elemi adatokra és elemi gépi kódú utasításokra bontotta fel. A strukturált szemlélet már az elemek szorosabb-gyengébb összetartozásának ábrázolására is rendelkezik eszközökkel. Az OO paradigma a gépközeli és a strukturált megközelítés szintéziseként önálló viselkedéssel rendelkező adatcsomagokból (struktúrákból) építi fel a számítógépes alkalmazást. Ezzel azonban a rendszert a gépközeli szemlélethez hasonlóan, többnyire függetlennek tekintett, egymással csak lazán kapcsolódó építőelemekből (stílusosan: kockákból) állítja össze.

használati, kapcsolódási módok

Ha az informatikai szemléletek fejlődése valóban egy spirális jelleget követ, akkor a következő paradigma ismét az építőelemek közötti viszonyokra, kapcsolódási módokra helyezi a hangsúlyt. Az objektumorientált megközelítés a rendszert szinte önálló, egymással lazán kapcsolódó objektumokra „robbantja" szét. Ez a felbontás azonban nem hogy megkönnyítené, hanem inkább megnehezíti, sőt, esetenként lehetetlenné is teszi, hogy az egyes építőelemek szerepét, a velük szemben támasztott követelményeket tisztázzuk. Az objektumokkal szemben támasztott igények ugyanis a határaikon kívül vannak, ezért azok csak a szükséges felhasználási — azaz kapcsolódási — módok ismeretében azonosíthatók.

A „vegytiszta" objektumorientált szemlélettől történő lassú elszakadás megfigyelhető volt már az OO technikák tényleges gyakorlati felhasználásának kezdetén, azaz a Smalltalk idejében is.

Kezdetben az objektumok, illetve az osztályok kapcsolatainak ábrázolását szervezési hibának tartották. Az akkori szemlélet szerint az objektum „magánügye", hogy kikkel van kapcsolatban, ezért ennek ábrázolása megsérti a bezárás *(encapsulation)* elvét. Néhány korai OO módszert, például az OMT-t is részben az tette népszerűvé és használhatóvá, hogy a technikákat kibővítették, szintetizálták a strukturált rendszerfejlesztési módszerek egyed-kapcsolat *(ER: entity-relationship)* jelöléseivel, például az asszociációkkal. Ezzel megkönnyítették a rendszerfejlesztők átállását az újabb szemléletre, másrészt szemléletesebbé tették az alkalmazás belső felépítését, architektúráját.

OO+ER

Az objektumorientáltsággal összefonódva jelent meg a *CRC-kártyák* technikája, illetve a *követelmény alapú tervezés (RDD: Responsibility driven design)*. Ezek azonban mégsem tisztán OO szemléletet képviselnek. Az OO „*mit* és nem a *hogyan*" szlogenjének mintájára a követelmények fonalát követő módszerek a „*minek* és nem a *mit*" elvére épülnek. Az RDD az objektumokat mintegy kívülről tekinti, hogy tisztázhassa azok rendszerben betöltött szerepét és így összegyűjthesse a velük szemben támasztott igényeket.

CRC-kártyák, RDD

Ugyancsak a tradicionális szemlélettől történő elszakadást jelent a Jacobson és társai által publikált *használati esetek* (use case) technikája, mellyel megkönnyítették az alkalmazással szemben támasztott követelmények összegyűjtését és formális ábrázolását. A használati esetek valójában a teljes alkalmazással szemben támasztott funkcionális követelményeket jelentik, ezért a módszer közeli rokonságban van az RDD-vel. A felhasználási lehetőségek vázlata ugyanis megegyezik az alkalmazás CRC-kártyájával.

használati esetek

A modern *komponens-architektúrák* esetén az objektumok belső tartalmáról a hangsúly fokozatosan áthelyeződik azok szolgáltatatásaira, azaz funkcióira. Egy komponens szerkezete, megvalósítási módja csak annyiban lényeges, amennyiben az kívülről elérhető és felhasználható. A komponens-architektúrák hozták meg az igazi áttörést a korábban többnyire csak elvi jelentőségű újrafelhasználásban, mivel ipari méretekben biztosítják a komponens-gyártás és felhasználás infrastruktúráját. Sajnos a legtöbb jelenlegi architektúra vagy kisipari méretű, vagy pedig csak egy érdekes kísérletként tekinthető, mivel nem biztosít „természetes" jelölést és közeget a komponensek definiálására és felhasználására. Egy mai, nagyipari méretű komponens-elvű alkalmazás ezért általában a megvalósítás szimulációs módszereinek mély ismeretét és többszörös többletmunkát igényel.

komponens architektúrák

„support"

A komponens-architektúrák egy érdekessége, hogy a „filozófiájuk" egy általános informatikai trendbe illeszkedik. Ez a trend *támogatásként* `(support)` fogalmazható meg, mely a speciális előtérbe helyeződését jelenti az univerzálissal szemben. Évtizedekkel ezelőtt egy általános rendszertől, egy programozási nyelvtől vagy éppen egy processzortól mindössze az univerzalitást követelték meg, például az legyen alkalmas bármely algoritmus megfogalmazására. Az azóta eltelt évek azonban megmutatták, hogy a gyakorlatban nem elegendő a megfogalmazhatóság elvi lehetősége, hanem a rendszereknek magát a megfogalmazást műveletét is támogatni kell. Ezért jelentek meg például a magasszintű, a strukturált és az objektumorientált programozási nyelvek. Az is erre az okra vezethető vissza, hogy a processzorokat kiegészítették a függvényhívás/visszatérést, illetve a többfeladatos működést megvalósító utasításokkal.

A támogatás az univerzalitást speciális keretek között valósítja meg. Az alprogram hívásának több lehetséges módja is van, az esetek elenyésző százalékától eltekintve azonban tökéletesen elegendő a mai processzorokba beépített utasítás. Az OO rendszerek késői kötés `(late binding)` technikájának, az adott művelethez tartozó metódus futási időben történő meghatározásának is több lehetséges módja van. A virtuális metódus tábla azonban ugyancsak egy kielégítő megoldást nyújt. Egy komponens-architektúra komponenseit is adott szabályok keretein belül kell definiálnunk és felhasználnunk.

„specializált közeg"

Egy specializált környezet egy *magasabb információtartalmú közeget* jelent. Mivel a környezetbe illesztendő elemeket adott szabályok betartásával kell megfogalmaznunk, ezért a környezet „többet tud" az elemekről és a több információ segítségével ki tudja egészíteni, *támogatni* tudja az elemek működését. A gyakorlat szempontjából megint csak lényeges, hogy a szabályokat a fejlesztőknek kell-e betartani, vagy egy „természetes" jelölésrendszerben megfogalmazott modellt valamely automatizmus a környezetnek megfelelővé tud alakítani. Első esetben a környezet előnyeinek elérése érdekében a fejlesztők a modell átfogalmazására, munkaigényes szimulációra kényszerülnek, míg a második esetben a támogatás a modellt „úgy, ahogy van" `(as is)` egészíti ki.

Java nyelv és környezet

A *Java nyelv és környezet* érdekesen illeszkedik az említett trendbe, ugyanis a legtöbb újdonsága pontosan a támogatással kapcsolatos. A Java virtuális gép (JVM) egy magasan specializált környezetet teremt meg, így például információja van a memóriaterületek felépítéséről és az objektumok adatmezőinek szerkezetéről. A Java képes azonosítani és felszabadítani a nem használt objektumokat `(garbage collection)`, így azt a támogatást nyújtja, hogy általában nem kell törődnünk az objektumok megszüntetésével. A specializált környezet több védelmi szolgáltatást is jelent, így külön kód írása nélkül is ellenőrzött az üres hivatkozások használata és a tömb-index határok figyelése, melyek a C/C++ programok egyik leginkább zavarba ejtő hibáit okozták.

A biztonságos programok írásának támogatása egészen a nyelv szintjéig terjed, mivel a Java-ból száműztek néhány olyan, a C/C++-ban létező konstrukciót, amely megnehezítheti a programok megértését. Az áttekinthető programnyelvi jelölések szempontjának sajnos áldozatul estek a generikus szerkezetek és az átdefiniálható operátorok is.

A Java „magasabb információtartalmú közeg" egy meglepő szolgáltatása a `reflection`, mellyel futás közben (!) feltérképezhetjük és használhatjuk rendszerünk objektumait. Ezzel a Java az interpreterek és a „szkriptelő" nyelvek képességeit és flexibilitását is a magáénak tudhatja.

A Java segítségével magas szinten valósítható meg az újrafelhasználás. Itt valóban építőkockákból építkezhetünk, mivel a már megírt osztályok egyszerűen beemelhetők a következő kifejlesztendő rendszerbe. A Java fejlesztés nem sokban tér el egy informatikai tudásbázis kialakításától, amivel egymásra épülő, egyre intelligensebb elemeket alkothatunk. A „tudásbázisban" összeállíthatjuk azt az informatikai infrastruktúrát, amely a szakterületi modell megvalósításához szükséges. Így egyre rövidebbé válhatnak a fejlesztési ciklusok, ugyanakkor egyre jobb minőségű alkalmazásokat készíthetünk.

A nyilvánvaló előnyök ellenére a Java-ban történő fejlesztés több kérdőjelet is von maga után. A Java egy olyan kísérletnek tekinthető, amely az elviselhető hatékonyság és a specializált környezet előnyei közötti keskeny határvonalon egyensúlyozik. A virtuális gép belső automatizmusait a processzorok általában nem támogatják, így a Java-val szemben az egyik leggyakoribb probléma a szimuláció miatt az igényelt erőforrások méretével és a hatékonysággal kapcsolatban merül fel. A virtuális gép részben elszakad a működtető környezettől, ezért a számítógép, sőt, az operációs rendszer bizonyos szolgáltatásai egyszerűen el sem érhetők.

A Java-jellegű szemléletet elsajátítva, azt viszonylag könnyen alkalmazhatjuk egy C++-os fejlesztés során. Az üzleti alkalmazások szempontjából a Java ugyanis nem rendelkezik megdöbbentő újdonságokkal. A környezet a megfelelő osztálykönyvtárakkal, a biztonságosság támogatása figyelemmel, a tárfelszabadítás pedig az élettartam explicit szabályozásával, a destruktorokkal helyettesíthető. A fejlesztés során választhatjuk azt a stratégiát, hogy a feladat első iterációját a produktív és biztonságos Java nyelven valósítjuk meg, majd azt átírjuk C++-ra.

A hatékonyság és a magas szintű támogatás közötti ingadozásra több példát is találunk. A Java rendelkezik egy `Java Beans` nevű komponens-architektúrával. Azonban a virtuális gép gyakorlatilag egyetlen speciális, kifejezetten a komponensek miatt beépített tulajdonsággal sem rendelkezik, holott ilyen lehetne például a tulajdonságok változásának automatikus figyelése, stb. Így az architektúra nem lényegesen kényelmesebb és produktívabb a vetélytársaknál. Ennél is meglepőbb, hogy maga a nyelv sem rendelkezik a komponensek írását és használatát megkönnyítő elemekkel, holott ez néhány kiegészítéssel, például a `class` helyén használható `bean` kulcsszóval megoldható lenne. A „komponens-architektúra" tehát egy utólagos hozzátoldás, a teljes rendszerbe szervesen nem illeszkedő kiegészítés, mely használata alapvetően a programsorok szabályoknak megfelelő feldúsítását, azaz szimulációt igényel. Hasonlóan nem illeszkedik szervesen a rendszerbe a távoli metódushívás (RMI), az adatbázisban történő tárolás és általában a perzisztencia sem.

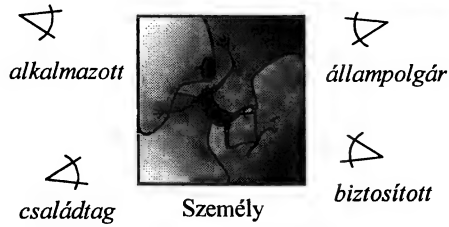
A Java és a C++ közötti harc még nem dőlt el. A gyakorlat feladatai egyfajta hibridet, gépközeli hatékonysággal megvalósított magas szintű támogatást igényelnek, mely lehetséges, hogy elérhető, például egy Java processzor segítségével. A csak néhány éves múltra visszatekintő, de rohamléptekkel fejlődő Java a támogatások területén is tartogathat még igazi meglepetéseket.

Létezik-e informatika az objektumokon túl?

Az eddig említett környezetek és technikák az elfogadott OO eszköztár elemeit bővítve „mindössze" az objektumorientáltság határait feszegették, de nem léptek azon túl. Néhány kísérleti paradigma, tervezési és programozási módszer azonban már szakít az OO „hagyományos", önállóságában létező osztály-fogalmával. Ezekben a kísérletekben mind az a közös, hogy az osztályokat nem a fejlesztés adott pontján minden részletükben meghatározott, zárt egységeknek tekintik. Ehelyett az alkalmazásnak az objektumorientáltságban osztályoknak nevezett egységeit több, részben vagy egészben független elemekből építik fel, sőt, módosíthatják is a már beépített elemeket. Az elemekre még nem alakult ki egyöntetű elnevezés. Leggyakrabban a *szempont* `(aspect)`, a *szerep* `(role)`, a *személyjegy* `(personality)`, a *koncepció* `(concept)` és a *vonatkozás* `(concern)` kifejezéseket használják.

Az „összeillesztgetés" szükségességét azon felismeréssel indokolhatjuk, hogy egy entitással szemben általában több, egymással csak lazán vagy egyáltalán nem kapcsolódó követelményeket fogalmazunk meg. Az új megközelítések segítségével az entitás különböző szempontokból tekintett jellegzetesség-csoportjait függetlenül definiálhatjuk és azok összeillesztésének eredményeként áll végül össze az alkalmazásban megjelenő osztály.

168. Ábra: „Objektum-puzzle"



A jellegzetesség-csoportok megvalósításával az osztály objektuma különböző helyzetekben más és más szerepekben jelenhet meg. Az egyes szerepek az objektum határain kívül interfész-jellegűek, azaz az objektum teljesíti az interfészben rögzített jellegzetességeket, ezért elérhető, használható a szerep által meghatározott módon. A szerep azonban több egy interfésznél, mivel a műveletekhez működés is tartozhat. A működések általában egy-egy alapértelmezett tevékenységsorozatot írnak le, melyek gyakran az entitás más (esetenként más szerepekben is rögzített) jellegzetességeire hivatkoznak.

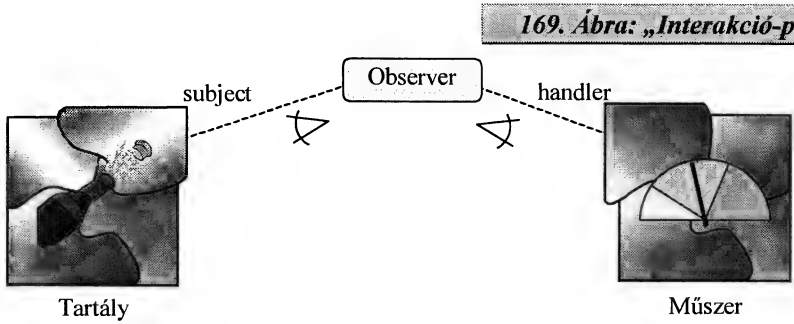
A szerepek egyszerű megoldási módot nyújtanak a rendszerfejlesztés egy régi problémájára. Gyakran szükségessé válik, hogy az objektumokat több különböző szempont alapján különböző öröklési hierarchiákba csoportosítsuk, korábbi példánk alapján egy személyt a neme, vagy a beosztása alapján. A szerepek segítségével az öröklést a nem változó jegyek alapján adhatjuk meg, az osztályokat kiegészíthetjük a lehetséges szerepekkel, illetve a szerepek esetén külön öröklési hierarchiákat is definiálhatunk.

A szerep és a kiindulási osztály nem általánosítás/pontosítás viszonyban van egymással. Egy „általános" személy nem általánosítása, sem nem pontosítása egy igazgatónak. Ehelyett egy konkrét személy adott szituációkban viselkedhet egy szerepének megfelelően, azaz elő tudja magát állítani az adott formába, önmagát „konvertálni" tudja a megfelelő szerepbe. Az átalakítást a ***Q*** módszer alapján a megfelelő művelettel végezhetjük el (pl.: `asIgazgató()`). Az azonban igaz, hogy a kiindulási osztálynak a szerep jellegzetességeivel történő kiegészítése az eredeti osztály pontosítása lesz. Az OO modellezéstől eltérő ez a szemlélet, de a technikákat részben ábrázolhatjuk a rendelkezésre álló eszközökkel (interfészek, realizáció, az UML «type» osztályai, stb.).

A szerepek nem csak statikusak, hanem dinamikusak is lehetnek, így például egy `Személyhez` hozzárendelhetjük, illetve arról később leválaszthatjuk egy `biztosított` jellegzetességeit. Ennek egy közbülső megvalósítási módja, ha minden entitásra felsoroljuk az összes lehetséges szerepét. Egyes rendszerek (például a `Personalities/J`) azonban azt is lehetővé teszik, hogy egy objektumhoz működés közben egy statikusan nem megadott szerepet is hozzárendeljünk. Sajnos ez a technika a jelenlegi futtató környezetekben csak igen körülményes szimulációval valósítható meg.

A szerep-jellegű elemekből építkező és az önálló osztályokra korlátozódó technikák közül az egyik legegységesebb és legszemléletesebb a `Luis Blando` nevéhez kapcsolódó *Személyjegyek alapján történő tervezés és programozás (`Design and Programming with Personalities`)*. Egy személyjegy leírhat olyan összetett tevékenységeket, amelyek végső soron bizonyos alsó-szintű művelet-csonkokra (az un. *`down-stream interface`*–re) hivatkoznak. Egy jármű adott útvonalat követő útja hivatkozhat például az `elindulás`, `előre haladás`, `balra fordulás`, `jobbra fordulás`, `megállás`, stb. elemi műveletekre. Ezzel valójában az összetett tevékenység aktivitás-diagramját fogalmazzuk meg, így lehetőségünk van arra, hogy a műveleteket az absztrakt modell szintjén írjuk le. A rendszerben így bármi alkalmas az útvonal követésére, ami megvalósítja ezeket az elemi műveleteket. A technika arra is ad lehetőséget, hogy egy személyjegy felvételekor annak elemi műveleteit és az alaposztály alkalmas metódusait összerendeljük. A személyjegyeken alapuló módszer egyetlen kérdéses pontja, hogy a nyelvbe a biztonságos tervezés érdekében túl sok korlátozást épít be. Az elemi művelet-csonkok kizárólag absztraktak lehetnek (így nem adhatunk meg alapértelmezett működést), az elérhető műveleteket nem definiálhatjuk felül, stb.

Az „objektumokon túli" tervezési és programozási módszereket kutató egyes irányzatok már nem csak egyetlen osztály összeállítását teszik lehetővé. Objektum-orientált rendszerszervezés esetén egy több osztályt is érintő vonatkozás beépítése azt igényli, hogy az osztályok jellegzetességeit kiegészítsük, feldúsítsuk olyan elemekkel, melyekkel az osztályok alkalmassá válnak a kívánt feladatra. Egy tartály és a telítettségét jelző műszer összekapcsolását megvalósíthatjuk például az `Observer` („Megfigyelő") tervezési minta segítségével. A két osztály összekapcsolása új jellegzetességek, attribútumok és műveletek beépítését igényli, illetve gyakran a már létező műveletek kiegészítését, esetleg módosítását is.



A több objektumot érintő vonatkozások megadására is alkalmas technikák közül az egyik legletisztultabb és legszemléletesebb a XEROX kutató központjában (PARC) folyamatosan fejlesztés alatt álló *Szempont-orientált programozás* (`AOP: Aspect-oriented programming`). Az elvek kutatásával párhuzamosan egy kísérleti AO környezetet, az `Aspect/J`-t is fejlesztik, amellyel a szemlélet alkalmazási lehetőségeit a gyakorlatban is tesztelhetjük.

Az AO technikákkal, illetve a Java környezetet előfordítóként (preprocesszorként) kiegészítő `Aspect/J`-vel az osztályokra épülő alapszerkezethez különböző, az osztályoktól függetlenül definiált *szempontokat* (`aspect`) adhatunk.

Egy szempont a már létező osztályok esetén *bevezethet* (`introduce`) új jellegzetességeket, azaz attribútumokat és műveleteket. Például egy raktárban a `mennyiség` és `minimális mennyiség` attribútumokkal rendelkező `Készlet` osztályt szeretnénk kiegészíteni olyan módon, hogy ismerje a készlet fogyásának intenzitását. Ezt úgy oldjuk meg, hogy az egyes vételezések esetén külön számoljuk, hogy hányszor maradt elegendő vagy csökkent a minimális alá a készlet. Ha az utóbbi viszonylag gyakori, akkor az áruból vagy gyakrabban, vagy nagyobb mennyiséget kell rendelni. Mivel az intenzitás mérésének attribútumai és műveletei a `Készlet` osztályban külön csoportot alkotnának, ezért azt először célszerű külön osztályként definiálni:

```
public class Intenzitás {
  private final double alapérték=0.5;
  private int normál, kivételes;
  public void clear() {    normál=kivételes=0; }
  public boolean notAvailable() { return (normál==0); }
  public double getValue() {
    return !notAvailable() ?
      (double)kivételes/normál:alapérték;
  }
  public void set(boolean f){if(f) kivételes++; normál++;}
}
```

Ezután fel kell vennünk a `Készlet` osztályban egy `Intenzitás` típusú attribútumot. Az Aspect/J-ben az alaposztálynak ezt a kiegészítését megtehetjük egy mindkét osztálytól független szempont megadásával:

```
aspect FogyásIntenzitás {
  introduce private Intenzitás Készlet.intenzitás
      =new Intenzitás();
  introduce public Intenzitás Készlet.getIntenzitás() {
    return intenzitás;
  }
}
```

Publikálással a Készlet osztályba átemelhetjük az intenzitás publikus műveleteit is, így a szempontba a `clearIntenzitás`, stb. műveleteket is felvehetjük.

A mérés helyes működése azonban azt is igényli, hogy minden vételezést jelentsünk az intenzitás objektumnak, illetve megadjuk, ha a készlet a kivételes esetnek számító minimális érték alá esett. Ehhez a `vételezés` művelet végén meg kell hívnunk az intenzitás beállító műveletét, így a szempontot a következő módon egészítjük ki:

```
advise static after * Készlet.vételezés(*) {
  intenzitás.set(mennyiség<=minimálisMennyiség);
}
```

Példánkban a `Készlet` osztály összes, vételezés nevű, de tetszőleges paraméterű és visszatérési-érték típusú műveleteinek végét kiegészítjük az intenzitás beállításának utasításával. Az `after` kulcsszót `before` kulcsszóra cserélve a vizsgálat a vételezés előtti állapotra vonatkozik.

Az `Aspect/J` nyelv előfordítója, az *aspect weaver*, amely „összeszövi" a különállóan megadott osztályokat és a szempontokat. A kiindulásként szolgáló osztályokat az `introduce` szerkezetek segítségével egészíthetjük ki újabb jellegzetességekkel. Az `advise` szerkezetekkel a már létező metódusokat módosítjuk, mely kiegészítő utasítások beillesztését jelenti a metódus elején (`before` kulcsszó), végén (`after` kulcsszó), valamint elfoghatjuk a fellépő kivételt (`catch` kulcsszó) vagy kiegészíthetjük a metódus végén, a kivételek esetén is végrehajtandó, zárás-jellegű tevékenységeket (`finally`).

A szempontok a programban osztályként jelennek meg, így külön attribútumaik és műveleteik is lehetnek, sőt, a szempontok is pontosíthatók. Például megtehetjük, hogy az összes készlet esetén is javasóljuk, hogy az utánrendeléseket gyakoribbá vagy ritkábbá kell tenni. Ehhez a `Fogyás intenzitás` szempontot a következőképpen kell pontosítanunk:

```
aspect EgyüttesÉsRészletes extends FogyásIntenzitás {
  static Intenzitás együttesIntenzitás =new Intenzitás();

  advise static before * Készlet.vételezés(*) {
    együttesIntenzitás.set(mennyiség<=minimálisMennyiség);
  }
}
```

Ha a példánkban nem alkalmazunk öröklést, akkor a „szövés" során mindkét szempontot meg kell adnunk. Öröklés esetén ez nem szükséges, mivel a pontosított szempont az ős definícióival is rendelkezik.

A „javaslatok" `(advise)` nem csak statikusak lehetnek. Mivel a szempontok a végső „összeszőtt" programban osztályként jelennek meg, ezért azok példányosíthatók, pl.: `new EgyüttesIntenzitás()`. A konkretizált szempont egyben bizonyos objektumok csoportját is jelenti, és ebben az esetben a nem-statikus javaslatok csak a csoportot jelentő szempontra vonatkoznak. Előző példánkat folytatva megtehetjük, hogy a fogyás részletes és együttes intenzitásának figyelése mellett, termékcsoportokat alakítsunk ki és azok fogyását külön is vizsgáljuk.

```
aspect CsoportIntenzitás {
  Intenzitás csoportIntenzitás =new Intenzitás();
  advise before * Készlet.vételezés(*) {
    csoportIntenzitás.set(mennyiség<=minimálisMennyiség);
  }
}
```

A szempont példányosítását, valamint egy készlet felvételét a csoportba a következő definícióval, illetve utasítással végezhetjük el:

```
CsoportIntenzitás csoport = new CsoportIntenzitás();
csoport.addObject( készlet )
```

Ezután az adott csoportokba tartozó `készlet` objektum `vételezés` művelete esetén a megfelelő csoportok intenzitása (is) változik.

A szempont-orientált programozás, illetve a hasonló eszközkészletre épülő technikák nagy előnye, hogy az objektumok közötti funkcionális kapcsolatokat, a közöttük lezajló interakciót az objektumoktól függetlenül is képesek definiálni. Ez hasonló minőségi ugrást jelent, mint amikor az eredetileg bezártnak tekintett objektumok közötti strukturális kapcsolatokat a modellezés során kiemelték és az objektumoktól függetlenül ábrázolták. Az asszociációk jelölése azonban csak az egyed-kapcsolat (ER) modellezés során már bevált technika feleleventését jelenti, míg az AOP és társai egy új, még felfedezetlen úton járnak.

Egymással funkcionális kapcsolatban álló osztályok bizonyos jellegzetességei esetén nehéz döntenünk, hogy azokat mely osztályoknál adjuk meg. Ez gyakran azért van, mert a jellegzetességek sem egyik, sem másik osztályhoz nem tartoznak, hanem magára a kapcsolatra vonatkoznak. Az új megközelítések alkalmazásával ezek ténylegesen a kapcsolathoz rendelhetők, például a szempontokat is tekinthetjük funkcionális kapcsoló-elemeknek, hasonlóan az asszociációs-osztályokhoz.

Talán ilyen rövid bemutatásból is érzékelhető, hogy az említett valószínűsíthető informatikai trendben a strukturális viszonyok háttérbe szorulnak, a funkcionális kapcsolatok pedig előtérbe kerülnek. Egy utolsó pillanatra igénybe véve az Olvasó türelmét: képileg ez a négyszög (az osztályokat jelölő téglalapok) körösítését (mely a funkciókra utal) jelenti.

Irodalomjegyzék

Objektumorientált szemlélet és programozás — magyar nyelven:

- Angster E. *Az objektumorientált tervezés és programozás alapjai.* Saját kiadás, 1997.
- Kondorosi K., László Z., Szirmay-Kalos L. *Objektum-orientált szoftverfejlesztés.* ComputerBooks, 1997.
- Nyékiné Gaizler J. (szerk.). *JAVA 1.1 útikalauz programozóknak.* ELTE TTK Hallgatói Alapítvány, 1997.
- Vég Cs., Juhász I. *Objektumorientált világ.* IQSoft Rt., 1998.

UML:

- G. Booch, I. Jacobson, J. Rumbaugh. *OMG Unified Modeling Language Specification (draft).* www.rational.com/uml, 1999. május.
- G. Booch, J. Rumbaugh, I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.
- J. Rumbaugh, I. Jacobson, G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.
- Jacobson, G. Booch, J. Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.
- M. Fowler, K. Scott. *UML Distilled: Applying the Standard Object Modeling Language.* Addison-Wesley, 1997.

Klasszikusok:

- J. Alger. *Secrets of the C++ Masters.* Academic Press, 1995.
- G. Booch. *Object-Oriented Analysis and Design with Applcations. 2nd ed.* Benjamin Cummings, 1994.
- P. Coad, E. Yourdon. *Object-Oriented Analysis. 2nd ed.* Yourdon Press/Prentice Hall, 1991.
- P. Coad, E. Yourdon. *Object-Oriented Design.* Yourdon Press/Prentice Hall, 1991.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns.* Addison-Wesley, 1994.
- A. Goldberg, D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
- J. Gosling, K. Arnold. *The Java Programming Language, 2nd ed.* Addison-Wesley, 1998.

- I. Graham. *Object-Oriented Methods.* Addison-Wesley, 1994.
- I. Jacobson, M. Christerson, O. Jonsson, G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, 1992.
- R. Jones, R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management.* John Wiley & Sons, 1996.
- B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.
- Object Management Group. *Object-Oriented Analysis and Design Reference Model.* Object Management Group, 1993.
- J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.
- B. Stroustrup. The C++ Programming Language. Addison-Wesley, 1986.

„Beyond Objects":

- L. Blando. *Designing and Programming with Personalities. MS Thesis.* Northeastern University (TR# NU-CCS-98-12), December 1998. (http://www.ccs.neu.edu/home/lblando/personalities)
- W. Harrison, H. Ossher. *Subject-Oriented Programming.* In Proceedings of OOPSLA 1993. Sigplan Notices, Vol. 28, No. 10, pp. 411-428, 1993. (http://www.research.ibm.com/sop/)
- G. Kiczales. *Aspect-Oriented programming.* ACM Computing Surveys, 28A(4), Dec. 1996. (http://www.parc.xerox.com/spl/projects/aop/)
- K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, 1996. (http://www.ccs.neu.edu/research/demeter/biblio/dem-book.html)
- Vég Cs. *Az OO szemlélet architektúrális támogatása.* II. Objektumorientált Országos konferencia, 1996 okt. (http://www.westel.hu/~logos2k/exp/, HU P9700007 pending).
- Vég Cs. *Objektumorientált alkalmazásfejlesztési módszer.* III. Objektum-orientált Országos konferencia, 1997 nov. (www.dragon.klte.hu/~vega/q/)

Tárgymutató

A

abstract class, 129
abstract operation, 129
abstract state, 173
absztrakció, 4
absztrakt állapot, 173, 175, 182, 183
absztrakt esemény, 176
absztrakt megoldási mód, 10, 222
absztrakt művelet, 129
absztrakt osztály, 129, 132
action, 143, 178, 181
action state, 167, 181
active, 143, 148, 153
activity, 143, 164, 168, 179, 181
activity diagram, 164
activity state, 168, 181
actor, 38, 39, 188
adatfolyam, 187
adatfolyam-diagram, 187
adatszolgáltatási felelősség, 90
adattár, 188
aggregáció, 113
aggregate value, 187
aggregation, 113
akció, 143, 178, 181
akció-állapot, 167, 181
aktív művelet, 97
aktív objektum, 148, 153, 201
aktív osztály, 143
aktivitás, 164, 165
aktivitás-állapot, 167, 168
aktivitás-diagram, 164, 167, 168, 173
aktor, 38, 40, 42, 56, 76, 188
alapértelmezett konstruktor, 136
algebrai szabály, 210, 211
alkalmazási diagram, 216
alkalmazási környezet, 221, 222
alkalmazási példák, 61, 62, 68
állapot, 145, 167, 173, 175, 179, 180, 194, 201
állapot-átmenet diagram, 164, 168, 173, 174, 182
alosztály, 120, 121, 123, 124, 196
általánosítás, 42, 46, 54, 119, 122, 182
analízis, 2
analysis, 2, 13
ancestor, 119
anchor, 192
AOP - Aspect Oriented Programming, 26
association, 99, 146
association class, 112
aszinkron üzenet, 154, 156, 159, 176
asszociáció, 99, 102, 104, 105, 113, 131, 146, 207, 224
asszociációs osztály, 112, 207
átfedő pontosítás, 122, 124
átmenet, 165, 174, 175, 183, 200, 201
áttekintés, 32, 33, 68
attribute, 77
attribútum, 77, 78, 81, 86, 90, 94, 98, 101, 114, 136, 145, 167, 195, 206
attribútumértékek felajánlása, 113
automatic transition, 178
automatikus átmenet, 178, 183

B

backward reference, 98, 192
backward service, 192
beállító metódus, 81
behavior, 78
belső akció, 171, 180, 181, 194
belső állapot, 173, 183
bemeneti akció, 168, 179, 194
bezárás, 80
bidirectional association, 103
bináris asszociáció, 99
binding, 127, 196
blob, 14
bonyolultság kezelése, 4
Booch módszer, 14, 17, 21
brain storm, 94
branch, 157

C

C++, 12, 238
cache, 79, 80, 224
cached, 84
call event, 176
cascading delete, 113
centre-out, 23, 61, 63, 70

change event, 177
class, 74, 121
class attribute, 138
class method, 138
classification, 74
collaboration diagram, 152
collection, 103
compartment, 87
composition, 114, 115
concrete class, 129
concrete state, 173
condition, 154, 178
conform, 121
connection, 217
constraint, 36, 141
construction, 135
constructor, 135
container class, 103, 132, 223
continuous loop, 179
continuous transformation, 186
control flow, 189
controller, 34
conversion, 137
copy constructor, 136
CRC-kártyák, 14, 93
csomag, 52, 56, 95, 194, 206, 208, 211
csomópont, 216

D

DAG - directed acyclic graph, 109, 128
data flow, 187
data flow diagram, 187
data store, 188
dealing with complexity, 4
decision, 166
default constructor, 136
default transition, 178
delegáció, 93, 113
delegation, 93, 113
dependency, 53, 117
deployment diagram, 216
derived association, 104
derived attribute, 80
derived class, 80
derived element, 80
descendent, 119
design, 2, 13
design by contract, 141
design pattern, 161
destructor, 135
destruktor, 135, 158, 194
dinamika, 164, 174
dinamikus osztályozás, 124
dinamikus típus, 198
discriminator, 123
disjoint generalization, 122, 124
diszjunkt pontosítás, 122
döntés, 166
dynamic classification, 124
dynamic typing, 198

E

early binding, 197
egész-rész, 113
egyed, 74
egy-életciklusú diagram, 179
egyesített osztály, 121, 122, 123, 197
egyetlen osztályozás, 122
egyirányú asszociáció, 103
együttműködési diagram, 152
elaboration, 225
elemzés, 2, 9, 13, 19, 21, 23, 24, 71, 80, 91, 95, 111, 118, 149, 200, 204, 205
elérés felelőssége, 98, 105
elérési útvonal, 54
életvonal, 155, 156
ellenőrzött, 23
előfeltétel, 141, 201, 210
előfordulás, 6
előzetes iteráció, 71
embedding, 198
encapsulation, 80
entitás, 74
entitás-objektum, 92
entity, 74
entry action, 179
enumeration, 92, 104, 115
érték-objektum, 92
érték-osztály, 115
értéktárolási felelősség, 91
értelmezés, 32
esemény, 67, 174, 175, 176, 177, 178, 180, 193, 200
eseményfolyam, 64
eseménygenerálás, 180
eszköztár, 139
event, 174, 176, 193
event-flow, 64
example object, 153
exception, 87, 176, 194
exit action, 179
extension, 125
extension point, 44, 87

F

feature, 86
feldolgozás, 186, 187
felelősség, 86, 87, 90, 93, 94, 96, 98, 105, 111
felfüggesztett állapot, 153, 156
felhasználói felület, 11, 65, 66, 68, 71, 76, 90, 93, 94, 95, 111, 112, 116, 140, 195, 205, 206, 208, 209, 210, 211, 223
felhasználói felületek diagramja, 68
feltétel, 67, 154, 166, 178
felügyelt művelet, 142
figyelő, 193
final state, 168, 178
finomítás, 118
flow-chart, 164
focus of control, 156
fogalomszótár, 32, 49
folyamatábra, 164
folytonos transzformáció, 186
foreign key, 223
forgatókönyv, 63, 64, 66, 67
fork, 187
függőség, 53, 54, 58, 117, 131, 146, 152

G

garbage collection, 11, 135
General System Science, 8
generalization, 119
generikus osztály, 107, 108, 132
generikus szerkezet, 107
getting method, 81
globális csomag, 54
globális függvény, 138
globális objektum, 146
globális változó, 138
glossary, 32
guard, 157, 178
GUI - Graphical User Interface, 11
gyorsítótár, 79, 80
gyűjtemény, 103

H

használati eset, 15, 16, 20, 21, 38, 40, 42, 43, 46, 51, 57, 90, 93, 94, 95, 152, 161, 194, 204
használati eset diagram, 38, 49, 55, 64, 95
hatványtípus, 117
helyzetjelentés, 193
hívás-esemény, 176
horgony, 192

I

időben lezajló változás, 78, 146, 164, 173, 174
idő-esemény, 177
implementáció és teszt, 25
implements, 130
inheritance, 119, 120, 196
initial state, 168, 178
inkrementális fejlesztés, 21, 23
instance, 6, 74, 138
instance attribute, 138
instance operation, 138
instantiation, 6, 138
integráció és együttes teszt, 25
interaction diagram, 152
interakció-diagram, 152, 161
interface, 129
interfész, 129, 130, 215
internal action, 171, 180
invariáns, 141, 210
invariant, 141
iterációs jelző, 154
iteration marker, 154
iteratív, 71
iteratív fejlesztés, 21, 23
iterator, 104
iterátor, 104

J

jellegzetesség, 86, 198
jelzés, 87, 154, 167, 171, 176, 193, 201
join class, 121, 197

K

kapcsolat, 98, 99, 146, 217
késői kötés, 197
készülék, 217
kétirányú asszociáció, 103
kezdőállapot, 168, 178
kibernetika, 8
kiemelés, 4, 121
kimeneti akció, 168, 179, 181, 194
kiterjesztés, 58, 125
kiterjesztett állapot, 173, 200
kivétel, 87, 142, 176, 194, 201
kizáró általánosítás, 123
kizáró pontosítás, 122, 123, 124
kollekció, 103, 108, 139, 147, 205, 223
komponens, 114, 214, 215, 216
kompozíció, 114, 115, 197, 198
kompozit állapot, 183

konkrét, 6
konkrét állapot, 173, 175
konkrét osztály, 129
konkurencia, 183
konkurens folyamat, 156
konkurens művelet, 143
konstans adattár, 188
konstrukció, 135, 158, 192
konstruktor, 135, 136, 137, 194
konverzió, 137, 209
korai kötés, 197
korlátozás, 125, 127
kötés, 127, 196, 197
követelményelemzés, 24, 70, 204, 210
közös sztereotípia, 87
közreműködő, 41
kulcsszavas érték, 34, 36, 37
külső állapot, 173

L

late binding, 197
láthatóság, 81, 138
layer, 208
leágazás, 154
leképezés, 10, 25
lekérdezés, 42, 79, 106, 143
lekérdező metódus, 81
lényegi tartalom, 52
leszármazott, 224
life cycle, 178
lifeline, 155
link, 98, 99, 115
listener, 193
lock, 143
lokális objektum, 147
lokális változó, 156
lollipop, 131, 215

M

magasabbrendű asszociáció, 99, 113
mapping, 10, 25, 224
másoló konstruktor, 136
megfigyelhető állapot, 79
megkülönböztető, 123, 124, 126
megosztás, 5
megszorítás, 34, 36, 80, 87, 91, 126, 141
megvalósít, 121, 130, 131
message, 153, 155, 176
message passing, 153, 155
metaclass, 140
metaosztály, 140
method, 78
method war, 15
metódus, 78, 129, 130, 196, 197
minősített asszociáció, 110, 111
minősítő, 110, 207
mirror, 195, 223
modell, 2, 8, 10, 19, 104
modellező nyelv, 6, 10, 17, 19, 20, 36
modifier, 79
módosító, 79, 94, 143
módszertan, 19
multiobject, 147, 153
multiobjektum, 147, 153
multiple classification, 122, 197
multiplicitás, 100
multiplicity, 100
munkafolyamat-diagram, 164
működési leírás, 65, 68, 206
művelet, 78, 79, 86, 95, 97, 98
művelet kiterjesztése, 127
művelet továbbítása, 113

N

name compartment, 75
namespace, 54
navigability, 103
névrész, 75
névterület, 54
node, 216
not null, 101
notify, 97
nyomkövetés, 118

O

object, 74, 144
object diagram, 144
object-flow, 170
Objective-C, 12
Objectory, 15, 16, 17, 20, 21, 34
objektum, 74, 144
objektum osztály, 75
objektum példány, 75
objektumazonosító, 75
objektum-folyam, 170
objektumorientált szemlélet, 7, 11, 19, 21, 74, 79, 113, 186, 192
observable state, 79
OCL - Object Constraint Language, 36
OID - object identifier, 75
OMG - Object Management Group, 15
OMT - Object Modeling Technique, 17, 19, 22, 100, 121, 123, 138, 220
one-shot diagram, 179

operation, 78, 181
operator overloading, 134
or-association, 111
osztály, 74, 92, 99, 101, 112, 211
osztályattribútum, 138
osztályművelet, 138, 146
osztályozás, 74
overlapping generalization, 122, 123
overloading, 78
overriding, 127

Ö

öndelegáció, 159
öröklés, 119, 196
őrszem, 157, 166, 178, 200
őrzött művelet, 142
ősosztály, 120, 121, 122, 123, 124, 137, 196, 224
összetett érték, 187, 188, 192
összhangban van, 120, 121

P

package, 52, 194
paraméter, 79, 83
parameterized class, 132
partíció, 208
partition, 208
passzív művelet, 97
passzív objektum, 148, 153, 188, 201
példány, 6, 74, 117, 144
példányképzés, 6, 138
példányosítás, 117
persistent, 37
polimorfizmus, 127, 196
polymorphism, 127
pontosítás, 54, 120, 121, 123, 124, 182, 196, 198
postcondition, 141
powertype, 117
precondition, 141
preliminary iteration, 71
privát, 81
private, 81, 117, 121
process, 19, 187
procesz, 19
processzor, 217
propagáció, 113
propagation, 113
protected, 81, 117, 192
pseudo state, 168, 178
pszeudo-állapot, 168, 178
public, 81
publikálás, 107, 209
publikus, 81

Q

qualified association, 110
qualifier, 110, 207
query, 79

R

RDD - responsibility-driven design, 14, 26
read only, 88
realize, 130
real-time, 156
reflexív asszociáció, 99
release, 225
rendezettség, 102
rendszerszervezés, 1, 2
requirement, 211
responsibility, 86, 211
restriction, 125
részekre bontás, 4
részfunkció, 45, 46
részletes tervezés, 24, 220, 222
réteg, 208
return, 154
reuse, 5, 121
role, 100, 102

S

sáv, 155, 168
scenario, 63
self-delegation, 157, 159
send, 176
sequence diagram, 152, 155
sequential, 142, 165
service class, 192, 195
setting method, 81
setup, 225
sharing, 5
signal, 88, 143, 167, 171, 176, 193, 201
signature, 78
SIMULA '67, 11, 12
single classification, 122
slot, 144
Smalltalk, 11, 237
start state, 168, 178
start-állapot, 168
state, 167, 173
state chart, 173
state-transition diagram, 173

static classification, 124
static typing, 198
statikus osztályozás, 124
stereotype, 34, 141
stick-note, 30
stop state, 168, 178
stop-állapot, 168
stratégiai tervezés, 24, 220, 221
strukturális kapcsolat, 99
subclass, 119
substitutability, 121
superclass, 119
swimlane, 155, 168
synchronization, 165, 166
synchronization bar, 165
szakterületi elemzés, 24, 71, 205
számosság, 100, 101, 112, 147
származtatás, 80, 84, 104, 105, 117
származtatott asszociáció, 104
származtatott attribútum, 80, 84, 88, 195, 200, 222, 224
származtatott osztály, 80
szekvencia-diagram, 152, 155, 156
szekvenciális, 142, 165
szerep, 100
szerepnév, 102, 103, 114
szignál, 143, 154, 176
szignatúra, 78
szinkronizáció, 160, 166
szinkronizációs vonal, 165, 166
szójegyzék, 32
szolgáltató osztály, 192, 193, 195
sztereotípia, 34, 35, 87, 141
szuperállapot, 182
szuperosztály, 119

T

tagged value, 36
tároló osztály, 103, 108, 131
teljes általánosítás, 120
template, 132
template class, 132
természetesebb jelölések, 8
ternáris asszociáció, 99, 103, 113
tervezés, 2, 9, 13, 19, 21, 23, 24, 80, 85, 91, 105, 109, 112, 118, 139, 149, 174, 195, 206, 207, 209, 211, 220, 222
tervezési minta, 26, 161
tevékenység, 62, 65, 143, 164, 168, 178, 179, 181, 204
tevékenység-állapot, 181
time event, 177
típuskonverzió, 137
több számosságú változat, 125
többalakúság, 196
többszörös osztályozás, 122, 128, 197
törlés továbbítása, 113
transition, 165, 174, 178
túlterhelés, 78
tükör-osztály, 195, 223

U

újrafelhasználás, 5, 121
unidirectional association, 103
Unified Method, 16, 20
use case, 15, 16, 38, 40
use case diagram, 38
user interface, 66
utility, 101, 139
utófeltétel, 141, 201, 210

Ü

üzenet, 87, 153, 176, 193, 201, 209, 223
üzenetküldés, 153, 155, 178, 180, 193

V

vagy-asszociáció, 111
változás-esemény, 177
változtatható, 88, 102
végtelen ciklusú diagram, 179
vezérlésfókusz, 155, 156
vezérlésfolyam, 189
virtuális metódus tábla, 196, 197
viselkedés, 78
visibility, 81
visszafelé hivatkozás, 98, 192
visszatérési érték, 79

W

whole/part, 113
work-flow diagram, 164

Z

záróállapot, 168, 178

Oktató Centrum
1142 Budapest
Teleki Blanka u. 15-17.
Tel: 363-2200
Fax: 220-5598

Az IQSOFT Oktató Centruma 1990 óta működik, 1995 óta az Oracle Hungary hivatalos oktató partnerei vagyunk. Tanfolyamainkon a legkorszerűbb technológiákkal és eszközökkel ismertetjük meg hallgatóinkat az alkalmazásfejlesztés, ügyvitelszervezés és a minőségbiztosítás területén.

Oktatóink saját szakembereink, akik az elméleti tudáson kívül az egyes területeken szerzett gyakorlati tapasztalataikat is átadják a hallgatóknak.

Igényesen kialakított oktatótermeinkben ideális körülményeket biztosítunk a tanfolyami résztvevőknek.

Ízelítő jelenlegi tanfolyami kínálatunkból:

- **UML a szabványos objektumorientált modellező nyelv**
- **Rational Rose és egyéb Rational eszközök**
- **Java programozási nyelv kezdőknek**
- **Java az elosztott rendszerek fejlesztésében**
- **Corba programozás Java és C++ környezetben**
- **Ügyviteli folyamatok professzionális szervezése - workflow a gyakorlatban**
- **Üzleti folyamatok modellezése (BPR)**
- **Projektirányítás... minőségi megközelítés**

Tanfolyamainkról bővebb információ található a www.iqsoft.hu címen.
Szóbeli információkkal Balogh Judit az oktatási osztály vezetője áll az érdeklődők rendelkezésére a 363-2200/185 telefonon vagy a baloghj@iqsoft.hu email címen.

Volt már, hogy elbizonytalanodott egy alkalmazás fejlesztésének közepén, a programsorok ezreibe gabalyodva? Szerette volna, hogy egy-két hét alatt a birtokába jussanak a modern technológiát képviselő szervezési és programozási módszerek, jelölések és technikák? Érezte már szükségét egy „guru" tanácsainak, aki javaslataival kisegíthetné, ötleteivel továbblendíthetné a fejlesztés megakadt menetét? Ön most éppen ezt a tanácsadót tartja a kezében. Nem kér különleges bánásmódot, sem napidíjat. Mindenkor a rendelkezésére áll, és ha éppen érdektelen vagy érthetetlen a mondandója, tudásának más oldalára kérdezhet rá.

Ajánljuk ezt a könyvet mindazoknak, akik már fejlesztettek, éppen fejlesztenek vagy fejleszteni fognak számítógépes alkalmazásokat és szeretnék azokat mind magasabb szinten, a jó értelemben vett nagyipari módon kivitelezni. Ajánljuk továbbá azoknak, akik érdeklődnek az informatika alkímiájának műhelytitkai iránt. Az alapismereteket tárgyaló részek és a technikai megoldások széles spektruma miatt a könyv egyszerre tankönyvként és a fejlesztés mindennapjainak kézikönyveként is alkalmazható.

A könyv, a szabványossá vált objektumorientált vizuális modellező nyelv, az UML (Unified Modeling Language) jelöléseit ismerteti, valamint egy olyan egyszerű, mégis hatékony módszert, amellyel alkalmazásainkat többéves tapasztalat nélkül is gyorsan kifejleszthetjük.

A tartalom három fő részre osztható. Az első részben a kifejlesztendő alkalmazással szemben támasztott igények, követelmények összegyűjtésére kapunk egy hatékony módszert és jelölésrendszert. Ez a rész nem hivatkozik objektumorientált technikákra, sem mélyebb informatikai ismeretekre. A második rész az UML legjobban kidolgozott elemét, az osztálydiagramot tárgyalja, amellyel az alkalmazás alapvető szerkezeti felépítését ábrázolhatjuk. Itt olyan módszert is találunk, amit követve a rögzített követelményekből félig automatikusan felépíthetjük az alkalmazás vázlatos modelljét. A harmadik nagyobb rész az UML további, kevésbé kidolgozott diagramjainak jelöléseit tekinti át – az alapvető jelentőségű osztálydiagramokhoz így szemléltetésként további diagramokat kapcsolhatunk. Itt megtalálunk egy olyan pontokba foglalt lépéssorozatot is, amit vezérfonalként használhatunk egy tipikus fejlesztés folyamatán.

Felépítése miatt a könyv ismeretanyaga az egyes részek elolvasásával párhuzamosan a gyakorlatban is alkalmazható.

„Az első igazán jó könyv, amit az UML-ről olvastam. Szemléletformáló és ötletgazdag. Szombaton reggel belenéztem, aztán vasárnap estig nem tudtam letenni."

Németh Miklós

ISBN 963 03 7660 1

9 789630 376600

Ára: 1950 Ft